Пояснювальна записка
до дипломного проекту
на тему: «Система автоматичного розгортання та аналізу продуктивності серверів»

Київ – 2019 рік

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API	Application Programming Interface – інтерфейс програмного додатку
CPU	Central Processor Unit – центральний процесор
GPU	Graphics Processing Unit – графічний процесор
RAM	Random Access Memory – оперативна пам’ять
GCP	Google Compute Platform
EC2	Elastic Compute Cloud – сервіс надання віртуальних машин від AWS
GCE	Google Compute Engine – серівс надання віртуальних машин від GCP
CSV	Comma-Separated Values – формат представлення табличних даних
PaaS	Platform as a Service – платформа як сервіс
SaaS	Software as a Service – програмне забезпечення як сервіс
IaC	Infrastructure as Code – інфраструктура як код
IoPs	Input/Output Operations Per Second – кількість операцій вводу-виводу в секунду
DNS	Domain Name System – система доменних імен
VPC	Virtual Private Cloud – віртуальна приватна хмара
VT	Virtualization Technology – технологія віртуалізації
SLA	Service Level Agreement – договір про рівень надання послуг
SLI	Service Level Indicator – показник рівню наданих
SLO	Service Level Objective – мета рівню наданих послуг
IaaS	Infrastructure as a Service – інфраструктура як сервіс

ЦОД	Центр Обробки даних
ПЗ	Програмне Забезпечення
ОС	Операційна Система

ВСТУП

Впродовж останніх п’ятнадцяти років, кількість операцій, проведених в публічних хмарних центрах обробки даних зростає на понад 20% щорічно [1]. Щорічно декілька тисяч нових стартапів обирають саме хмарні платформи в якості своїх обчислювальних центрів [2]. Все більше існуючих проектів мігрують до хмарних, так званих On-Premises (на потребу) дата-центрів, відмовляючись від власних серверних у своїх місцях розташування.

Така популярність обумовлюється тим, що оренда хмарних обчислювальних потужностей знімає з замовників величезний тягар відповідальності та потенційно можливих проблем: не потрібно піклуватися про стабільну лінію зв’язку зі стабільно високою швидкість з’єднання, стабільне енергопостачання, необхідність проходити сертифікації з безпеки та надійності, думати про надійність обчислювальних машин та закладати бюджет на їх оновлення. Одною з популярних причин також є модель оплати за послуги “Pay as you go” (оплачуй у процесі використання) [3]. Така модель дає можливість споживачу послуг провайдерів хмарних послуг платити лише за ті ресурси, які він споживає, а в разі необхідності- переключитися на більш (або менш) дорогі та продуктивні ресурси в залежності від конкретної потреби споживача.

Найбільші техно-гіганти, такі як Microsoft, Amazon, Google, Red Hat, IBM пропонують публічні хмарні дата-центри з неймовірно різноманітними сервісами, проте, серед них є дуже схожі. Наприклад, сервіси реляційних баз даних (CloudSQL, RDS), сервіси обчислень на віртуальних машинах (GCE, EC2, Cloud Compute), сервіси взаємодії через повідомлення/черги (SQS/SNS, Pub/Sub). Досить складно обрати саме ту

компанію, яка буде пропонувати найкращі технології з точки зору надійності, співвідношення ціна-якість, продуктивність, простота у використанні, наявність у більшості регіонів нашої планети.

Понад половина IT-ринку нині обирає саме хмарні обчислення [4], на відміну від створення власних дата-центрів. Досить складно обрати оптимального провайдера послуг. Створення системи, що може з легкістю розгорнути віртуальні сервери у декількох центрах обчислень та проведе оцінку їх продуктивності зможе дати можливість споживачам більш ретельно підходити до питання вибору провайдера послуг хмарних обчислень, що у майбутньому зможе зберегти досить великі гроші.

Створюване впродовж роботи над даним дипломним проектом ПЗ призначене для прискореного розгортання та аналітики продуктивності серверів у хмарних дата-центрах.

Метою проекту є економія часу, коштів та отримання більш точної картини щодо якості послуг хмарних провайдерів з мінімальними витратами коштів та часу.

Для досягнення поставленої мети необхідно проаналізувати існуючі хмарні провайдери, визначити алгоритм для оцінки їх продуктивності, виявлення переваг та недоліків, спроектувати, реалізувати та експериментально дослідити наявні на ринку представників хмарних обчислювальних послуг.

1 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ

На даний момент на ринку присутня велика кількість провайдерів послуг хмарних обчислень. Кожен з них пропонує велику кількість послуг, серед яких є як однотипні, наприклад, сервіс надання віртуальних машин, так і досить вузькоспеціалізовані такі, як сервіси нереляційних баз даних для вузько визначених потреб. Серед найбільших провайдерів таких послуг можна виділити Amazon Web Services, Microsoft Azure, Google Cloud Platform та Hetzner [5].

Управління ресурсами хмарних провайдерів може бути виключно мануальне чи ж автоматизоване з використанням різноманітних додатків. Це можуть бути додатки для конкретних представників послуг, наприклад, CloudFormation для Amazon, чи ж універсальні – Terraform.

Для конфігурації та тестування віртуальних серверів застосовують доволі різні підходи. Наприклад, cloud-config для віртуальних машин, чи ж маніпуляції з серверами опісля їх розгортання за допомогою Ansible, Chef, Puppet та інших.

1.1. Різновиди хмарних обчислень

Хмарні обчислення дуже тісно пов’язані з такими поняттями та технологіями як:

- Інфраструктура як сервіс (“Infrastructure as a Service” або “IaaS”);
- Платформа як сервіс (“Platform as a service” або “PaaS”);
- Програмне забезпечення як сервіс (“Software as a Service” або “SaaS”).

1.1.1 Інфраструктура як сервіс

Інфраструктура як сервіс (IaaS) – хмарний комплекс послуг, за допомогою якого надається доступ до обчислювальних ресурсів, наприклад, віртуальних або виділених серверів, сервісам зберігання даних, віртуальних мереж (також, файєрволи та інші захисні мережеві технології) [6]. Дане рішення є повноцінним аналогом створенню власного дата-центру. Провайдери таких послуг беруть на себе питання щодо конфігурації та керування системами віртуалізації, фізичними серверами, системами збереження даних та мережевим стеком.

Найчастіше, такий вид послуг є найдешевшим серед вище перелічених, адже провайдери таких послуг не розробляють власні сервіси, користувацькі бібліотеки для них та інше програмне забезпечення, а лише займаються питанням надійного енергопостачання, ефективного тепловідводу та забезпечення стабільного інтернет-з’єднання. Споживачу таких послуг зазвичай доступні декілька платформ (операційних систем) на вибір, що можуть бути встановлені на орендовані віртуальні (або виділені) сервери та можливість конфігурації отриманого серверу (обрання необхідної кількості ядер процесору, оперативної пам’яті та розміру дискового простору).

Основні функції:

- Замість придбання власного обладнання та необхідності відразу оплатити його вартість, користувачі оплачують IaaS в ході його використання;
- Інфраструктура легко масштабується в залежності від необхідності- кількості пам’яті та обчислювальної потужності;
- Дані зберігаються у хмарі, тому немає потенційної єдиної слабкої точки відмови;

- Відмова від необхідності підтримки задач віртуалізації звільняє час на виконання робіт інших типів.

Прикладом таких платформ є хостинг провайдер “Ukraine” (“Україна”) або ж німецький “Hetzner”. Очевидним вибором така послуга є в разі необхідності розширення існуючого власного дата-центру без можливості фізичного оновлення або розширення, або ж бажання отримати повноцінний контроль над ресурсами, що надаються, адже конфігурація отриманих віртуальних машин повністю лежить на відповідальності замовника.

Очевидним недоліком такого рішення є те, що найчастіше провайдер гарантує лише безвідмовну роботу самих серверів, а за встановлене програмне забезпечення та його надійність відповідальність несе лише споживач. В разі виходу з ладу програмного забезпечення, наприклад, помилка у роботі кластеру MySQL, що призвела до втрати даних, неможливо перекласти відповідальність на провайдера послуг, адже він відповідає лише за роботу серверу.

1.1.2 Платформа як сервіс

Платформа як сервіс (PaaS) – надає застосункам не лише інфраструктуру, а й готові інструменти для розробки й тестування власних розробок [7]. Разом з платформою як сервіс розробники отримують набір бібліотек та фреймворків для роботи, які можуть бути використані у кінцевому програмному забезпеченні.

Основні функції:

- PaaS надає платформу з інструментами для розробки, підтримки та тестування застосунків в тому ж середовищі;
- Команди розробників отримують змогу зосередитися на процесі розробки та оптимізації коду, а не підтримці інфраструктури;

- Спрощується командна робота, особливо якщо команда знаходиться у різних місцях розташування;
- Провайдери керують безпекою своїх продуктів;
- Зазвичай такі продукти легше масштабуються та підтримуються.

Вибір такої послуги значно спрощує процес розробки програмного забезпечення, його підтримку та розвиток. Також дуже часто такі рішення є більш витрато-ефективними, адже базові компоненти, наприклад, бази даних запускаються на спеціалізованих для таких цілей віртуальних машинах з набором необхідних оптимізацій. Також зазвичай провайдери таких послуг гарантують безперебійну роботу своїх продуктів, тому в разі їх відмови досить легко отримати грошову компенсацію або ж спеціальні умови, що є досить вагомим бонусом для великого бізнесу.

Провайдерами таких платформ є Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP). Серед їх продуктів є сервіси для реляційних баз даних: Relational Database Service (RDS) від Amazon, Azure SQL від Microsoft та CloudSQL від Google; сервіс взаємодії повідомленнями Pub/Sub від Google та SQS/SNS від Amazon. Або ж AWS Elastic Beanstalk, що надає операційну систему, оточення для виконання програм, написаних різними мовами програмування, базу даних, веб сервер та інше. Провайдери гарантують те, що їх сервіси будуть працювати без перебоїв у більш ніж 99,9% випадків, в іншому ж разі можливо отримати спеціальні умови користування сервісами або компенсації у грошовому еквіваленті. Для формалізації таких умов використовуються поняття Service Level Agreement (SLA), Service Level Objective (SLO) та Service Level Indicator (SLI) [8].

Слід зауважити, що досить часто PaaS провайдери пропонують й IaaS сервіси. Наприклад, у вищезгаданих представників ринку можна й орендувати різноманітні віртуальні машини за допомогою таких сервісів:

EC2 (Elastic Compute Cloud) від Amazon, Google Compute Engine від Google та Azure Compute від Microsoft.

Сервіс розгортання віртуальних машин від Amazon Web Services був запущений ще у далекому 2006 році. За більш ніж 13 років були змінені та кардинально перебудовані центри обробки даних, змінені покоління фізичних процесорів, їх архітектури, технології віртуалізації, підхід до забезпечення високонадійних систем.

На даний час сервіс Elastic Compute Cloud підтримує 2 типи віртуалізації для створення віртуальних машин:

- Паравіртуалізація (Para Virtualization - PV);
- Апаратна віртуалізація (Hardware-assisted Virtual Machine - HVM);

В свою чергу, Hetzner та Google Cloud використовують KVM (Kernel-based Virtual Machine).

Така гібридна модель надання обчислювальних можливостей на даний момент є найбільш популярною на ринку серед споживачів.

1.1.3 Програмне забезпечення як сервіс

Програмне забезпечення як сервіс – це модель роботи з хмарними провайдерами, що передбачає надання комплексного програмного рішення як сервіс. Доступ до таких рішень здійснюється через мережеве з’єднання шляхом використання браузеру або ж спеціальних клієнтів [9].

Основна перевага такого підходу є відсутня необхідність на підтримку таких платформ, їх розробку, оновлення та інше. Цільова аудиторія таких продуктів – кінцеві споживачі.

Основні функції:

- Можливість отримання доступу до програмного забезпечення через підписки;

- Відсутність необхідності інсталювати, обслуговувати та управляти програмним комплексом;
- Дані захищені провайдером послуг;
- Використання легко масштабується в залежності віл потреб користувача;
- Додатки досить часто доступні з будь-яких девайсів (мобільні телефони з різними платформами, комп'ютери з різними операційними системами).

Прикладом таких послуг є CircleCI, що надають платформу для тестування, збірки та інших етапів розробки програмного забезпечення. Також, іншим прикладом є Google з набором Documents, Sheets, Slides, що є повноцінним аналогом Microsoft Office у вебі. Відсутня необхідність купувати досить дорогу ліцензію, а можна лише отримати підписку на необхідний період часу за менші гроші, що є великим бонусом для користувачів. До того ж, непотрібно встановлювати програмне забезпечення локально, що дозволяє ще й економити дисковий простір на робочих машинах.

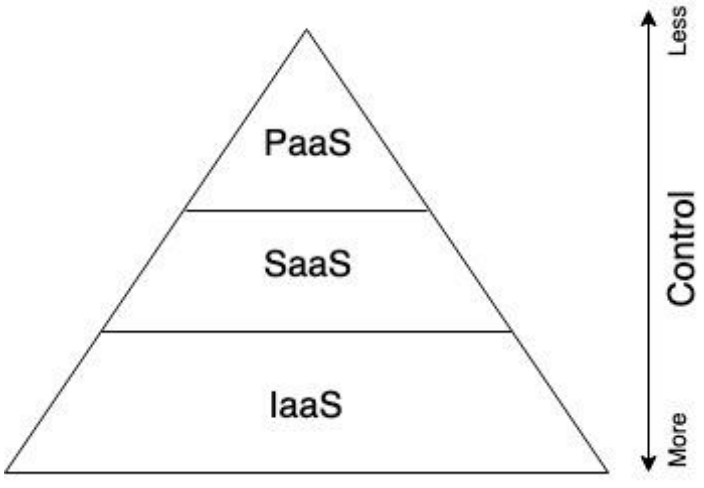

Рисунок 1.1 – Порівняння рівню контролю над платформою для різних типів послуг

1.2 Огляд існуючих рішень

На момент створення дипломного проекту не існує комплексного рішення, що б дало змогу досягти поставленої мети у повному обсязі.

Наявні лише поодинокі таблиці порівнянь продуктивності тих чи інших сервісів окремих провайдерів хмарних послуг. Серед спільних недоліків існуючих рішень:

- Застаріла інформація/неактуальні типи віртуальних машин;
- Тести проведені на окремих платформах та неможливо провести порівняння з іншими платформами, адже тестування не є універсальним та автоматизованим;
- Відсутність широкого вибору програмних засобів, що використовувалися при тестуванні.

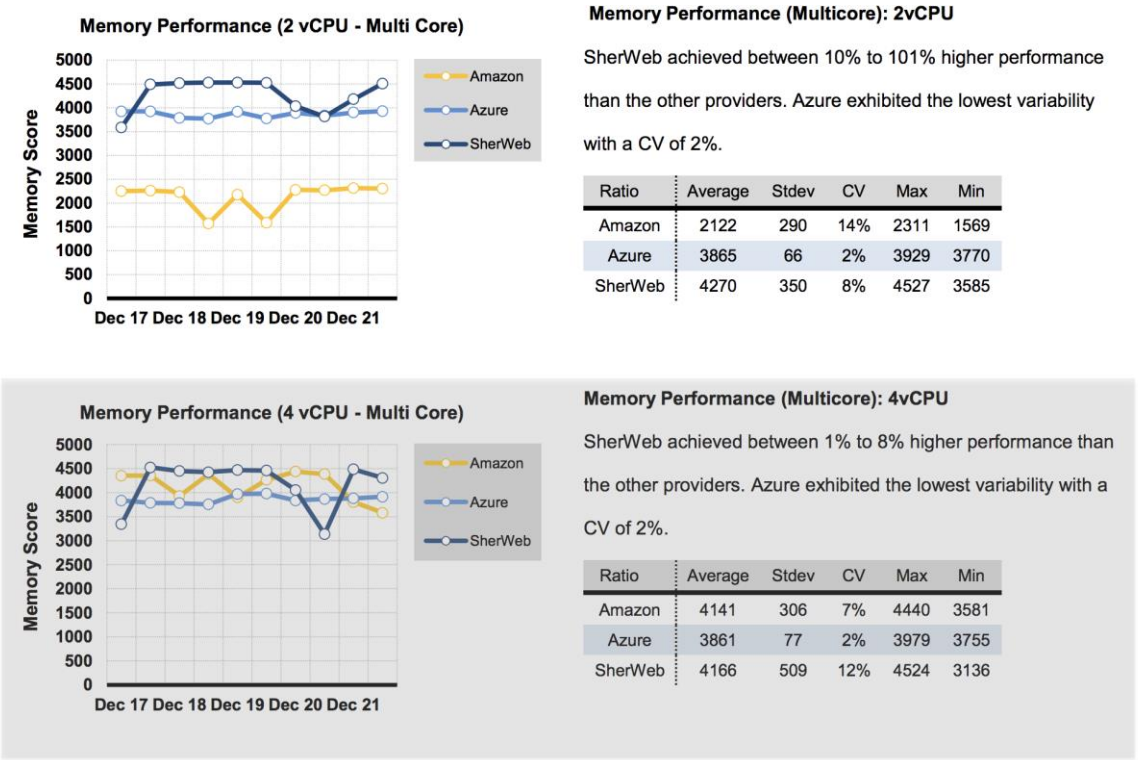

Рисунок 1.2 – Приклад результатів тестування від CloudSpectator.com

1.3 Опис найпопулярніших гіпервізорів

Основою будь-якого центру обробки даних є віртуалізація. Незалежно від того, йде мова про власно-розгорнуті датацентри (так звані on-site) або ж датацентри провайдерів хмарних послуг, дуже рідко використовують системи, коли операційна система встановлена напряму на апаратну частину. Технологія віртуалізації дозволяє встановлювати на одну й ту саму фізичну машину декілька повністю ізольованих систем.

Перші технології віртуалізації з'явилися ще у 1960-х роках, проте справжня необхідність у використанні таких технологій побачила світ лише наприкінці 20-го сторіччя через небувалий до того часу розвиток індустрії й стрімко зростаючому попиті на обчислювальні потужності й віртуальні машини [10]. Саме тоді актуальною стала потреба у повноцінному використанні всіх фізичних ресурсів, розгортанні програмного забезпечення, забезпечення надійності, безпеки та швидкого відновлення систем в разі виникнення непередбачених обставин у вигляді катастроф.

Незалежно від того, яка саме технологія віртуалізації розглядається, завжди існує машина фізичного рівня (хост) та програмне забезпечення (гіпервізор), що відповідає за розгортання та управління всіма іншими віртуальними машинами в рамках даного хосту.

Існує три види взаємодії віртуальних машин з фізичним хостом:

- Динамічна трансляція;
- Паравіртуалізація (PV);
- Апаратна віртуалізація (Hardware Virtual Machine).

Перший представник з переліченого списку не дає змоги дізнатися операційній системі, що вона є віртуальною. В даному випадку гіпервізор перехвачує всі системні виклики операційної системи за підмінює їх на більш безпечні. Такий підхід дещо сповільняє перехвачені команди.

Проте, такий підхід дозволяє віртуалізувати будь-які операційні системи, адже втручання у ядро систем не потребується.

Дана технологія активно використовується у продуктах компанії VMWare, яка є одним з лідерів ринку розробників комерційного програмного забезпечення для віртуалізації [11]. Проте, компанія має й широкий спектр продуктів для некомерційного використання.

На відміну від динамічної трансляції, паравіртуалізація потребує втручання до програмного коду гостьових операційних систем для того, щоб інструкції виконувалися більш безпечно та ефективно. При такому підході операційна система завжди знає про те, що система лише віртуальна, адже були внесені корективи у ядро. Через такий підхід неможливо віртуалізувати деякі операційні системи, зокрема Windows та OS X, адже їх програмний код не є відкритим [12].

Серед переваг такого підходу- покращена продуктивність, у порівнянні з динамічною трансляцією, а от серед недоліків- неможливість запуску віртуальних систем на базі Windows та OS X.

В свою чергу, апаратна віртуалізація є найновішою розробкою серед перелічених. Ще у часи розробки процесорної архітектури x86, її інженери розуміли, що дана архітектура погано підходить для віртуалізації, оскільки вона з початкових етапів розробки була спроектована для запуску лише однієї операційної системи. Після того, як на ринку з’явилися динамічна трансляція від VMWare та паравіртуалізація від Xen, найбільші виробники прцесорів на той час (й нині) – Intel та AMD почали випускати процесори з апаратною підтримкою віртуалізації.

На момент представлення технології, помітного приросту в продуктивності не було, адже спочатку більший фокус був на покращення архітектури процесорів. Проте, після понад 10 років активної роботи над даною технологією, слід зауважити, що нині апаратна віртуалізація у

вигляді таких рішень як VT-x від Intel та AMD-V від AMD нічим не поступається своїм аналогам та у деяких випадках навіть показує себе більш ефективним рішенням.

1.3.1 KVM

KVM (Kernel-based Virtual Machine) – програмний комплекс для віртуалізації, що є вбудованим в ядро Linux. Дана технологія нічим не поступається своїм конкурентам й навіть є кращою в деяких аспектах. Слід зауважити, що KVM – технологія з відкритим кодом, тому величезна кількість розробників з усього світу мають змогу покращувати її. Також, компанія Red Hat активно використовує дану технологію та впроваджує у своїх продуктах [13].

З початкових етапів розробки KVM, інженери були сфокусовані на підтримці апаратної віртуалізації. За великим рахунком, гіпервізор- невеликий програмний комплекс, або ж операційна система, котра має мати змогу керувати мережею, оперативною пам’яттю та дисковим простором. Цей функціонал повноцінно реалізований у ядрі Linux, тому цілком логічним є використання саме ядра в якості гіпервізора. Таким чином, кожна окрема віртуальна машина, створена з використанням даної технології- лише окремий процес в операційній системі Linux. Безпека забезпечується за допомогою SELinux/sVirt, а контроль над використанням ресурсів – за допомогою CGroups.

Нещодавно KVM став частиною ядра Linux та їх розвиток є паралельним. В процесі роботи KVM напряму звертається до процесора через специфічний модуль (в залежності від того, який процесор використовується на рівні хосту) – kvm-intel або kvm-amd. В основі KVM лежить апаратна віртуалізація HVM.

KVM є основною технологією для віртуальних машин центру обробки даних Hetzner. Загалом, це єдина послуга даного IaaS провайдеру.

Також KVM лежить в основі сервісу надання віртуальних машин Google Compute Engine. Вони використовують так звану “Вкладену Віртуалізацію” (“Nested Virtualization”). Загалом, це встановлення KVM гіпервізорів (L0) на фізичні машині, а в межах кожного гостьового (L1) домену встановлюється ще один KVM гіпервізор, який в свою чергу вже керую віртуальними машинами, котрі доступні клієнтам даного сервісу.

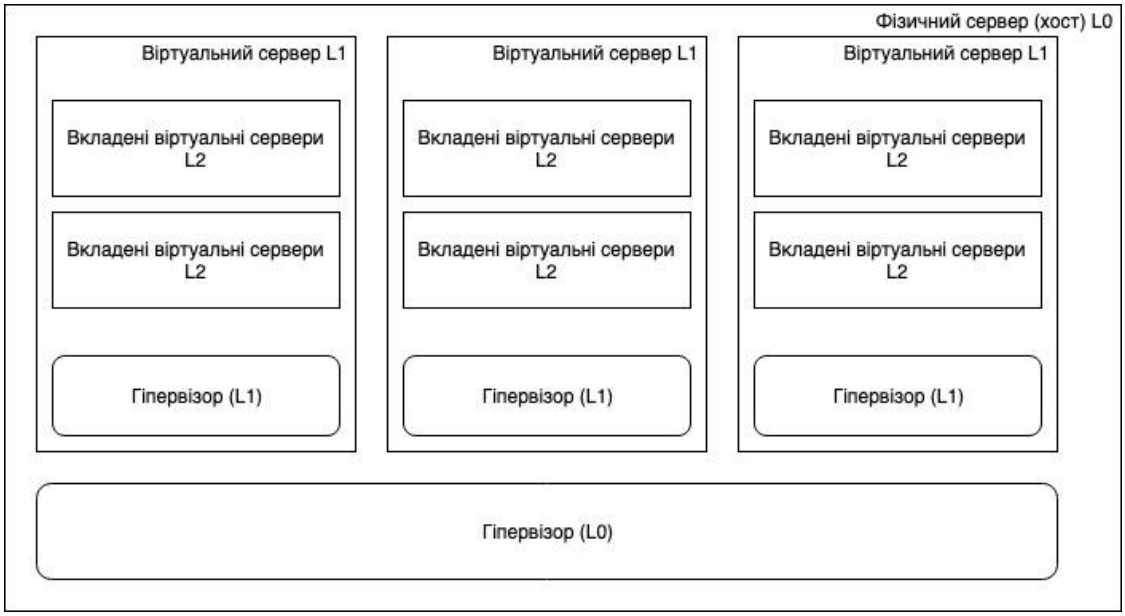

Рисунок 1.3 – Схематичне зображення серверів створених за технологією вкладеної віртуалізації

Серед обмежень такої технології слід виділити:

- Вкладена віртуалізація можлива лише для L1 віртуальних машин, створених на базі процесорів Intel архітектури Haswell та більш сучасних;
- Підтримуються лише KVM гіпервізори з використання лише операційної системи на базі Linux;

- Лише вкладені віртуальні машини (L2) можуть бути запущені з операційною системою Windows певних версій.

До переваг даної технології відносяться:

- Лише до 10% втрати продуктивності процесорів на L2 рівні вкладених віртуальних машин;
- Можливість легкого переносу віртуальних машин між фізичними серверами, що забезпечує легкість у масштабуванні та відтворенні систем.

1.3.2 Xen

Xen – один з найпопулярніших гіпервізорів, що підтримує одразу 2 технології віртуалізації: апаратна (HVM) та паравіртуалізація (PV).

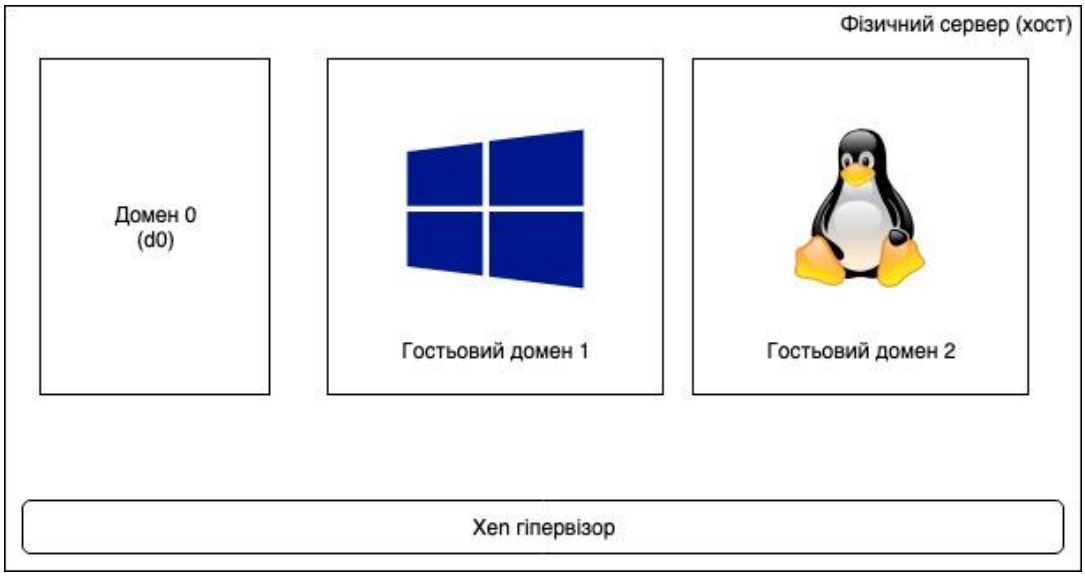

Рисунок 1.4 – Схематичне зображення компонентів Xen гіпервізора

Кожна віртуальна машина створена Xen гіпервізором називається гостьовим доменом. Існує й нульовий домен, котрий виконує функцію управління іншими гостьовими доменами. Більшість функцій, пов’язаних

з віртуалізацією перенесені саме в домени, таким чином Xen гіпервізор є найменшим з своїх конкурентів, проте домени в такому випадку виконують більше системних низькорівневих функцій.

Домени є повністю віртуалізованими, тому такі операційні системи не знають про те, що на справді вони ділять процесорний час з іншими гостьовими доменами. Такий спосіб реалізації віртуалізації дозволяє запускати будь-які операційні системи без жодних змін у своєму програмному коді.

Xen гіпервізор дозволяє використовувати як технологію HVM, так і PV. PV є дещо швидшою, проте потребує певних змін у ядрі системи. Досить часто системи, що запускаються за допомогою PV є модифікованими дистрибутивами Linux [14].

Нині саме Xen гіпервізор використовується у сервісі Amazon Web Services – Elastic Compute Cloud (EC2). Одним з головних недоліків є те, що у кожному регіоні (центрі обробки даних у певному географічному регіоні) можливо використовувати персоналізовані для цього регіону образи віртуальних машин за умови використання технології PV через те, що у певних регіонах фізичні хост-машини можуть мати технологічні особливості. Дана технологія вносить певні обмеження щодо використання снапшотів віртуальних машин при їх перенесенні між різновидними фізичними серверами. Незважаючи на те, що PV є більш швидкою технологією віртуалізації для Xen, AWS все ж рекомендує використовувати саме HVM віртуалізацію через наступні причини:

- Покращена підтримка HVM у Xen гіпервізорі;
- Нові покоління процесорів з новітніми процесорними інструкціями;
- Покращення у EC2 драйвері, спрямованому на прискорення роботи HVM.

Більш того, для всіх сучасних типів віртуальних машин AWS EC2 рекомендовано використовувати саме HVM віртуалізацію.

AWS Nitro – технологія віртуалізації, що була представлена у 2017 році. В основі лежить KVM, проте велика кількість компонентів з KVM не використовуються, наприклад, QEMU, domO. Покращена система передачі переривань дозволяє збільшити продуктивність серверів, що приближує її майже до власне фізичних машин. Нове (5-те) покоління віртуальних машин від AWS лише на 1% повільніше за аналогічні фізичні сервери [15].

1.3.3 Docker

Docker – програмне забезпечення з підтримкою контейнеризації для розгортання та підтримки програмного забезпечення. На відміну від повноцінної віртуалізації, контейнери не є повноцінними віртуальними машинами. Вони лише додають шар ізоляції в рамках однієї операційної системи, а не емулюють її повнцінно. Такими заходами ізоляції можуть бути додавання унікального ідентифікатора контейнеру до кожного процесу або ж контроль дозволених системних викликів.

Таким чином контейнери можна вважати додатковим шаром ізоляції процесів, окрім існуючого контролю на основі користувачів та користувацьких груп у Linux системах. У випадках, коли подібні механізми недоступні, наприклад, при використанні операційних систем сімейства Windows, при розгортанні контейнеру Docker запускається віртуальна машина VirtualBox, що і забезпечує створення контейнеру. До речі, схожий механізм роботи був і у операційних системах OS X до нещодавнього оновлення систем даного сімейства [16].

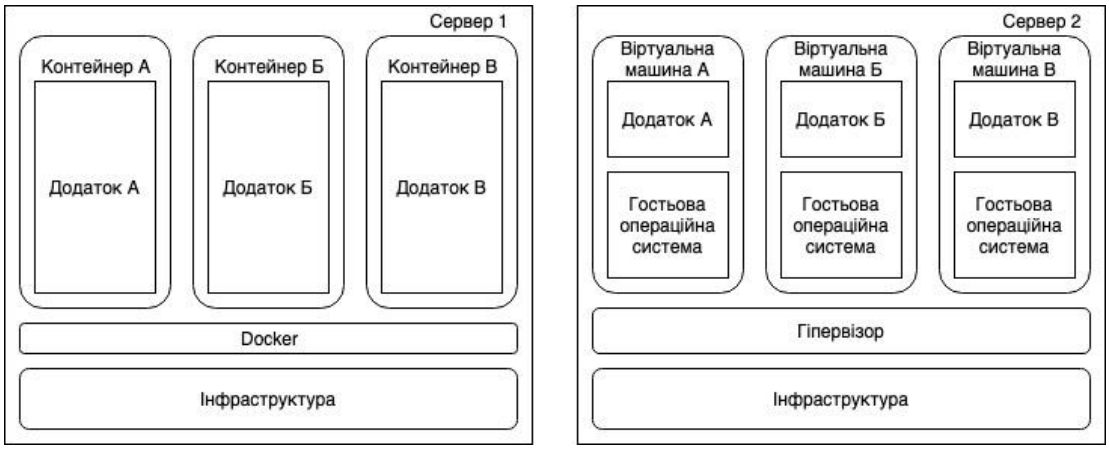

Рисунок 1.5 – Порівняння контейнеризації та віртуалізації

Слід зауважити, що нині технологія контейнеризації є надзвичайно популярною та за статистикою, більш ніж 25% комерційних проектів вже мігрували або планують міграцію на контейнери [17]. Дане рішення дозволяє надзвичайно легко масштабувати системи, їх розгортати у різних середовищах та навіть їх відтворювати. За її допомогою операційні інженери з легкістю можуть відтворити оточення в якому виконується той чи інший програмний код з метою тестування або повторення оточень для різноманітних демо презентацій або ж для детального відлаштовування.

Разом з технологією контейнеризації активно розвиваються з контейнерні оркестратори. Серед найпопулярніших – Elastic Container Service (ECS) від Amazon, Kubernetes від Google, Docker Swarm від Docker, Nomad від Hashicorp [18].

Одним з найпопулярніших оркестраторів нині є Kubernetes. Дана технологія є аналогом давно використовуваною в Google системи Borg. Нещодавно Google опублікувала в інтернеті програмний код Kubernetes і він набрав неймовірної популярності. Дана система здатна контролювати кластери з тисяч віртуальних машин, на яких розгорнуті десятки тисяч контейнерів.

1.4 Висновки до розділу 1

У процесі виконання першого розділу дипломного проекту було розглянуто основні типи послуг, що надаються провайдерами хмарних обчислень – Iaas, PaaS, SaaS.

Серед розглянутої класифікації видів послуг при використання IaaS користувач отримує повний контроль над наданими ресурсами, проте провайдер гарантує лише стабільну роботу віртуальних машин, мережі та дисків, але не несе відповідальності за встановлене власноруч ПЗ.

У разі використання послуг виду PaaS, користувачеві надаються платформи для створення програмного забезпечення. В такому випадку користувач не має доступу до інфраструктури наданої платформи, а її управлінням займається провайдер хмарних послуг. Окрім того, найпопулярніші провайдери гарантують надійність наданих систем. Вартість такого виду послуг вища, ніж власноруч створеної інфраструктури та розгорнутого ПЗ, проте інколи досить важно досягти навіть близьких до заявлених значень SLA. Досить часто провайдери послуг PaaS також надають послуги виду IaaS. Наприклад, AWS серед своїх продуктів має EC2, який є IaaS, хоча більшість продуктів AWS є PaaS послугами.

Послуги типу SaaS надають користувачеві доступ до повноцінного програмного забезпечення, не даючи змоги управляти конфігурацією та інфраструктурою даного ПЗ. Такий вид послуг є найдорожчим серед розглянутим.

Також було розглянуто способи автоматизації взаємодії з ресурсами хмарних провайдерів та найбільш популярні нині гіпервізори для створення віртуальних машин. Була встановлена відмінність між віртуалізацією та контейнеризацію. Враховуючи те, що фізичні процесори, що використовуються у дата-центрах хмарних провайдерів є

інколи однаковими, розглянуті відмінності у використовуваних гіпервізорах Xen/KVM у майбутньому дадуть змогу зробити висновки щодо ефективності їх використання тими чи іншими провайдерами хмарних послуг, адже AWS все ще досить багато використовує Xen, в той час як більшість конкурентів є прихильниками KVM.

2 АНАЛІЗ ТА ВИБІР ЗАСОБІВ АВТОМАТИЗАЦІЇ РОЗГОРТАННЯ ТА АНАЛІЗУ ПРОДУКТИВНОСТІ СЕРВЕРІВ

При обранні технологічних засобів для досягнення поставленої мети слід звернути увагу на такі загальні параметри:

- Простота у використанні та конфігурації;
- Можливість реалізації модульної архітектури;
- Розповсюдження під безкоштовною публічною ліцензією без обмежень.

Також слід зауважити, що важливим є й активна розробка та підтримка обраних програмних продуктів, їх популярність у спільноті та частото виходу нових версій, адже провайдери хмарних послуг розвивають свої продукти надзвичайно швидко та оновлюють своє API ледь не щодня, що іноді призводить до проблем у роботі засобів автоматизації.

2.1 Огляд засобів конфігурування ресурсів хмарних платформ

До застосунків для конфігурування інфраструктури важливими є наступні вимоги:

- Можливість використання інструменту з найбільшою можливою кількістю хмарних провайдерів;
- Коректна підтримка актуальних API хмарних провайдерів;
- Стабільна робота при поганому мережевому з’єднанні при створенні ресурсів з великими затримками.

2.1.1 Terraform

Terraform – продукт компанії Hashicorp, що спеціалізується на створенні інструментів для захисту, розгортання, забезпечення безпеки та оркестрації для інфраструктури додатків різних типів. Створений на мові програмування Golang. Серед продуктів даної компанії є надзвичайно популярні програмні комплекси такі як Vault, Nomad, Consul та Terraform. Компанія розробляє власну декларативну мову програмування HCL (Hashicorp configuration language) [19].

Дане рішення дозволяє створювати, змінювати, версіонувати та оперувати інфраструктурними компонентами, що є доступними в більшості найпопулярніших провайдерів хмарних обчислювальних сервісів. Даний інструментарій має такий рівень абстракції як провайдер. Зазвичай, провайдери є проектами з відкритим кодом, тому кожен може вносити необхідні зміни. Провайдер – обгортка над API провайдера ресурсів, якими провайдер має оперувати. Наразі у мережі доступні провайдери для таких IaaS як AWS, GCP, Azure, Hetzner, таких PaaS як Heroku та SaaS DNSimple, CloudFlare.

Даний інструментарій є втіленням підходу Infrastructure as a Code (IaC) [20]. Описання інфраструктури кодом надає чітку картину того, що в даний момент часу розгорнуто в дата-центрі, дає можливість швидко масштабувати ресурси та відновлювати їх в разі збою. Впродовж останніх кількох років даний підхід вважається революційним у сфері конфігурації хмарних провайдерів.

Terraform дозволяє розгорнути необхідні інфраструктурні ресурси для застосунків та провести їх базову конфігурацію. Основними напрямками використання є:

- Налаштування PaaS для веб-застосунків;

- Конфігурація інфраструктури для застосунків з декількома шарами розділення функціоналу (multi-tier app);
- Конфігурація власноруч розгорнутих кластерів (Nomad, ECS, Kubernetes);
- Розгортання оточень для застосунків на потребу (наприклад, створення демо-серверів, що мають повністю повторити конфігурацію вже існуючих серверів);
- Підтримка оточення застосунків у декількох хмарних дата-центрах.

Основною перевагою даного застосунку є те, що він є безкоштовним, з відкритим кодом, активно розвивається та підтримується спільнотою користувачів, а розробники беруть до уваги коментарі та виправляють знайдені користувачами помилки. Також даний інструмент дозволяє керувати численними хмарними провайдери та дозволяє створювати власні провайдери для власних дата-центрів.

2.1.2 CloudFormation

Даний застосунок створений компанією Amazon для керування ресурсами хмарних дата-центрів AWS. Він є універсальним інструменту для керування конфігурацією, автоматичного моделювання та безпечного управління будь-якими видами ресурсів, що представлені в AWS.

Також продукт дозволяє виконувати базову конфігурацію серверів, а не лише на рівні ресурсів хмарного провайдеру послуг.

CloudFormation використовує YAML для описання ресурсів, що інколи є досить незручним для читання, особливо у випадках коли інфраструктура є досить об’ємною. Розповсюджується даний застосунок безкоштовно, проте активно розробляється лише розробниками з Amazon. Він не набув великої популярності з часу свого виходу.

Проте, даний продукт активно розвивається та компанія AWS вкладає багато зусиль у розвиток продукту.

Серед недоліків даного продукту слід виділити наступні:

- Підтримка лише одного провайдеру хмарних послуг (AWS);
- Досить складні у сприйнятті YAML конфігураційні шаблони при великих обсягах;
- Код застосунку не є публічним.

2.1.3 Ansible

Ansible – система керування конфігураціями, що створена мовою програмування Python та використовує декларативну мову розмітки YAML для керування конфігураціями. Зазвичай використовується для управління Linux вузлами, проте Windows підтримується також. Окрім досить широкого спектру конфігурування систем, Ansible також дозволяє керувати хмарними ресурсами, наприклад створювати віртуальні машини або інші ресурси у різноманітних хмарних дата-центрах.

Ansible виконує описані у YAML-плейбуках директиви через SSH з’єднання з віддаленими серверами, що є досить безпечним та надійним способом комунікації.

Даний продукт є досить популярним, адже вже давно виступає на ринку. Близькими аналогами є Chef та Puppet. Нині актуальною є 3-тя версія застосунку. Спільнота користувачів Ansible досить велика, тому завжди легко знайти відповідь на свої запитання, що є безперечною перевагою перед конкурентами. Нещодавно корпорація Red Hat придбала даний продукт, тому слід чекати велику кількість покращень в найближчий час [21].

Слід зазначити, що для виконання команд через Ansible необхідно встановити Python на віддаленому сервері, щоб всі команди коректно виконувалися через інтерфейс утиліти.

Головними перевагами даного застосунку є:

- Простота у використанні;
- Можливість виконання команд на віддалених машинах без створення додаткових допоміжних серверів;
- Активна розробка та велика кількість робочих прикладів;
- Розповсюдження під безкоштовною публічною ліцензією.

2.2 Огляд засобів тестування продуктивності серверів

Існує досить велика кількість засобів тестування продуктивності серверів. Протестувати можливо всі основні параметри віртуальних машин: швидкість оперативної пам’яті, дисків для зберігання даних, мережевих карт, процесорів та навіть графічних процесорів.

Зазвичай, тести продуктивності центрального процесору полягають у виконанні складних математичних операцій, архівування та розархівуванні даних, шифруванні або дешифруванні. Такі операції можуть виконуватися як в однопоточному режимі (з використанням лише 1 ядра процесору), так і у багатопоточному (з використанням декількох ядер центрального процесору).

- Коректна робота програмних застосунків з сучасними процесорними інструкціями;
- Можливість роботи у багатопоточнноому та однопоточному режимах;
- Підтримка більшості актуальних ОС.

2.2.1 Stress-ng

Це програмний додаток, що створений для оцінки продуктивності різноманітних підсистем комп'ютеру. Він відноситься до класу внутрішніх тестів, адже у ході виконання програми не виконується жодних викликів до сторонніх сервісів. Утиліта функціонує цілком ізольовано. Серед наявних тестів є тести CPU, RAM, I/O. Оскільки у даному дипломному проекті проводиться саме тестування центральних процесорів, то саме запуск з вказання кількості доступних ядер процесору та таймаутом є найраціональнішим використанням даного пакету. Цей навантажувальний тест чудово працює з декількома процесорними потоками, тому саме він обраний в якості утиліти для тестування у багатопоточному режимі.

Для генерації навантаження даний тест виконує математичні операції, процеси стиснення та шифрування/дешифрування даних.

```
vtatarin@Valeriis-MacBook-Pro ~/P/deeploma> stress-ng --cpu 1 --timeout 30 --metrics-brief
stress-ng: info:  [43944] dispatching hogs: 1 cpu
stress-ng: info:  [43944] successful run completed in 30.03s
stress-ng: info:  [43944] stressor       bogo ops real time  usr time  sys time   bogo ops/s   bogo ops/s
stress-ng: info:  [43944]                          (secs)    (secs)    (secs)   (real time) (usr+sys time)
stress-ng: info:  [43944] cpu                8709     30.03     29.40      0.08       289.99       295.42
```

Рисунок 2.1 – Приклад результатів виконання тесту stress-ng

Перевагами даного застосунку є:

- Можливість проведення стрес-тестів для багатьох компонентів серверів (CPU, RAM, GPU, сховище даних, мережеві інтерфейси);
- Підтримка сучасних процесорних інструкцій;
- Можливість роботи у однопоточному та багатопоточному режимах.

Серед виводу результатів даного тесту найважливішими є поля "bogo ops/s" ("bogus operations per second"). Дана метрика показує

кількість операцій, що були виконані стрес тестом у одиницю часу (секунду). Саме цей показник можна вважати метрикою проудктивності машини, на котрій був запущений тест [22].

2.2.2 SuperPI

Одним з найпопулярніших тестів продуктивності центрального процесору є тест SuperPI. Це комп’ютерна програма, що виконує обчислення числа Пі з точністю до 32-х мільйонів знаків після коми. При цьому використовується алгоритм Гауса-Лежандра. З початку розробки програма була однопоточною, тому було досить складно оцінити продуктивність сучасних багатоядерних процесорів. Проте, останні модифікації дозволяють проводити обчислення й у багатопоточних режимах.

Одним з аналогів SuperPI є wPrime. Це більш сучасний аналог, що дозволяє виконувати обчислення за допомогою того самого математичного методу, проте операції обчислюються у декілька потоків. Також серед відмінностей є те, що програма містить алгоритми для вивантаження результатів тестування до глобальної таблиці результатів у мережі Інтернет [23].

Незважаючи на наявність більш розвинених та сучасних аналогів, SuperPI є одним з широковідомих та найбільш популярних тестів продуктивності серверів у однопоточному режимі. Особливої популярності даний засіб тестування набув серед оверклокерів. Оверклокінг (від англ. Overclocking) – підвищення швидкодії компонентів комп'ютера за рахунок експлуатації їх у форсованих (нештатних) режимах роботи.

Важливими перевагами даного застосунку є такі параметри:

- Наявність широкої бази результатів тестувань різноманітних систем;
- Простота у використанні;
- Підтримка ОС Windows.

```
vtatarin@Valeriis-MacBook-Pro ~/P/deeploma> docker run fewlaps/superpi -t 1 -i 32000000
Test started with 1 threads and 32000000 iterations
Test finished in 354 ms
```

Рисунок 2.2 – Приклад результатів виконання тесту SuperPI

До недоліків даного додатку слід віднести лише можливість проведення навантажувальних тестів лише для центрального процесору.

Серед виводу даного стрес тесту можна побачити таку інформацію як кількість задіяних потоків процесору для виконання математичних операцій, кількість ітерацій (операцій обчислення, що дорівнює кількості знаків після коми у обчисленому числі) та час (у мілісекундах), що був затрачений на виконання всіх ітерацій обчислення.

Саме метрика витраченого часу є найбільш важливою та буде використовуватися для порівняння продуктивності віртуальних машин під час проведення даного тесту.

2.2.3 Висновки до розділу 2

Серед проаналізованих засобів для проведення навантажувальних тестів для аналізу продуктивності центрального процесору були оглянуті найпопулярніші додатки – stress-ng та SuperPI. Останній додаток є досить популярним й серед широкої спільноти прихильників модифікації

власних персональних комп’ютерів з метою підвищення їх продуктивності.

Слід зазначити, що серед засобів автоматизації розгортання серверів найбільш ефективним є використання Terraform, оскільки він підтримує роботу з великою кількістю хмарних провайдерів та технологій, має відкритий код та активно розвивається.

Для маніпуляцій з віддаленими серверами найбільш раціональним є використання додатку Ansible, який, на відміну від конкурентів — Puppet та Chef, не потребує розгортання додаткових серверів для роботи, а лише встановлений на комп’ютері оператора Python та власне додаток. Також YAML синтаксис, що використовується для Ansible є простим у читанні та використанні, що робить навіть нетривіальні конфігурації легко зрозумілими.

Отже, враховуючи перелічені переваги та недоліки розглянутих утиліт, було для розробки системи було обрано такий стек компонентів: Terraform та Ansible для розгортання та конфігурації віртуальних машин, Stress-ng та SuperPI для проведення навантажувальних тестів продуктивності.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Опис задачі

Для забезпечення підтримки різноманітних публічних хмарних провадйерів необхідно реалізувати систему з модульною структурою, що дозволить незалежно оперувати разноманітними API хмарних провайдерів. Слід створити найбільш універсальне робоче оточення на розгорнутих серверах, що дозволить у подальшому їх використовувати для задач різних типів.

Одним з таким рішень є встановлення операційної системи CoreOS, що містить встановлений за замовчуванням Docker демон, що дозволяє у майбутньому з легкістю розгортати будь-які оточення для програмних додатків.

Модульна архітектура може бути забезпечена стандартними засобами додатку Terraform та Ansible.

Отримані результати тестування слід агрегувати у зручних для обробки формат даних. Наприклад, таким форматом може бути обраний CSV. Даний формат є текстовим форматом зберігання даних, є зручним для читання. Окрім того, дані у CSV форматі легко імпортувати для побудови порівняльних графіків за домогою різноманітних інструментів (Microsoft Word, LibreOffice, Pyplot).

Систему для порівняння продуктивності та конфігурацію слід провести для віртуальних машин різних типів у найпопулярніших хмарних ЦОД розглянутих раніше – Google Cloud Platform, Amazon Web та Hetzner. Зокрема, розроблені модулі мають підьримувати роботу з такими сервісами перелічених хмарних провайдерів як EC2, GCE та Hetzner Cloud.

3.2 Методика розгортання серверів

Розгортання віртуальних машин здійснюється за допомогою інструменту Terraform. За допомогою декларативної мови програмування HCL (Hashicorp Configuration Language) описується бажаний стан інфраструктури. Terraform транслює отриманий синтаксис у команди для провайдерів.

Провайдер – абстракція над API хмарного провайдеру послуг, що надає єдиний інтерфейс для взаємодії. Досить часто провайдери є продуктами з відкритим програмним кодом, тому спільнота має змогу вносити корективи та розширювати можливості програмних комплексів.

Оголошення роботи з провайдером виконується за допомогою блоку “provider”. В тілі блоку передаються необхідні параметри для ініціалізації роботи з API хмарного провайдеру послуг. Приклади ініціалізації провайдерів для Amazon Web Services, Google Cloud та Hetzner:

```
terraform {
  backend "s3" {
    bucket = "terraform-states-vtatarin"
    key    = "vm-performance/aws"
    region = "us-east-1"
  }
}

provider "aws" {
  region = "us-east-1"
}

provider "google" {
  project = "${local.service}"
  region  = "us-east4"
}

provider "hcloud" {}
```

Рисунок 3.1 – Приклад ініціалізації провайдерів Terraform

Всі дії, що виконує Terraform, записуються до файлу, що відповідає поточній переданій конфігурації програмного коду. Таким чином, Terraform має змогу знаходити відмінності від того, що наявне у коді від потенційно можливих внесених змін до конфігурації інфраструктури через маніпуляції з API напряму або ж через веб-додаток. Даний файл носить назву “terraform.tfstate” та зазвичай має зберігатися не локально на комп’ютері кожного розробника, а у одному з сервісів віддаленого зберігання даних (Amazon Simple Storage Service – S3, Google Cloud Storage – GCS). Файл має структуру json-об’єкту. Саме такий текстовий формат зберігання об’єктів вважається найбільш простим у використанні та читанні людьми, адже має просту ієрархічну структуру. Файл стану може містити секретні дані (ключі доступу до віддалених ресурсів), тому не рекомендовано зберігати цей файл у незахищеному форматі у системах контролю версій.

Можливо використовувати спеціально створений для цього сервіс – Terraform Remote State Storage, проте він є більш дорогим у використанні, аніж вище згадані аналоги. Конфігурація віддаленого зберігання стану інфраструктури може бути виконана наступним чином:

```
terraform {
  backend "s3" {
    bucket = "terraform-states-vtatarin"
    key    = "vm-performance/aws"
    region = "us-east-1"
  }
}
```

Рисунок 3.2 – Приклад конфігурації віддаленого зберігання стану Terraform

Існування сконфігурованого віддаленого місця зберігання файлу стану дозволяє також запобігти одночасному внесенню змін до одних й тих самих ресурсів. Такі одночасні виклики API хмарних провайдерів не завжди оброблюються належним чином та можуть призвести до конфліктів. Для цього для кожного tfstate файлу динамічно при звертанні до нього може бути згенерований tflock файл. Наявність такого файлу сигналізує про те, що до файлу стану в даний момент часу вже ведуться виклики, тому одночасні виклики для його зміни заборонені, поки не завершиться попередня сесія.

```
resource "aws_spot_instance_request" "spot_experiment" {
  count         = "${length(var.instance_type) * "${var.type == "spot" ? 1 : 0}"}"
  ami           = "${data.aws_ami.experiment.id}"
  instance_type = "${var.instance_type[count.index]}"

  key_name = "${var.admin_key_name}"

  user_data = "${data.ignition_config.default.*.rendered[count.index]}"

  subnet_id                   = "${element(random_shuffle.subnet.*.result[count.index], 0)}"
  associate_public_ip_address = true

  vpc_security_group_ids = [
    "${aws_security_group.default.id}",
  ]

  root_block_device {
    volume_type           = "${var.root_volume_type}"
    volume_size           = "${var.root_volume_size}"
    delete_on_termination = true
  }

  tags {
    Name    = "${var.service}-${var.instance_type[count.index]}"
    Service = "${var.service}"
    Spot    = "true"
  }

  spot_type              = "${var.spot_type}"
  block_duration_minutes = "${var.duration_minutes}"

  valid_from  = "${timestamp()}"
  valid_until = "${timeadd(timestamp(), "${var.timeout}")}"
}
```

Рисунок 3.3 – Приклад блоку HCL для створення віртуальної машини AWS EC2

Синтаксис HCL є досить простим у використанні. Наприклад, для створення віртуальної машини у Amazon Web Services Elastic Compute Cloud необхідно використати блок наведений на рисунку 3.3.

Однією з головних ідей використання Terraform є модульність. Так, створив модуль “EC2” що, наприклад, описує створення віртуальної машини у Amazon Web Services Elastic Compute Cloud, і містить у кореневій директорії наступні файли:

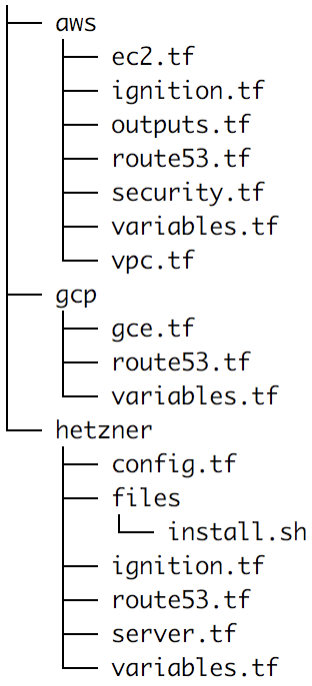

Рисунок 3.4 – Приклад директорії що описує модуль EC2

За допомогою стандартного функціоналу даного інструменту, можна об’явити створення екземпляру даного модулю, що може бути описаний одним з наступних способів:

- У локальній директорії;
- У віддаленій системі контролю версій (git);
- У Terraform Registry.

Остання опція доступна лише у бізнес версії даного інструменту. Лістинг з рисунку 3.5 описує створення описаних в модулі ресурсів з

параметрами, що є переданими параметрами service, type, vpc, admin_key та залишає решту змінних за замовчуванням:

```
locals {
  service = "vm-performance"
}

module "aws" {
  source = "../modules/aws"

  service = "${local.service}"
  type    = "on-demand"

  instance_type = [
    "t3.medium",
    "t3a.medium",
    "t2.medium",
    "c4.large",
    "m4.large",
    "m5a.large",
    "t2.large",
    "t3.large",
    "r4.large",
  ]
}
```

Рисунок 3.5 – Приклад виклику описаного у локальній директорії модулю

Для виконання даного коду вперше необхідно перейти до каталогу, що містить main.tf файл та виконати команду “terraform init” в командному рядку, що наведена на рисунку 3.6. Дана команда ініціалізацію описаних у файлах даної директорії модулів та провайдерів. Зокрема, виконується завантаження провайдерів, оновлення локального файлу стану ресурсів. Далі необхідно виконати “terraform apply” та відповісти “yes”, якщо користувач згоден зі змінами, котрі виведені та плануються бути виконані:

```
vtatarin@Valeriis-MacBook-Pro ~/D/d/v/test> terraform init
Initializing modules...
- module.aws
  Getting source "../modules/aws"
- module.gcp
  Getting source "../modules/gcp"
- module.hetzner
  Getting source "../modules/hetzner"

Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...
- Downloading plugin for provider "random" (2.1.2)...
- Downloading plugin for provider "aws" (2.14.0)...
- Downloading plugin for provider "null" (2.1.2)...
- Downloading plugin for provider "google" (2.8.0)...
- Downloading plugin for provider "hcloud" (1.10.0)...
- Downloading plugin for provider "ignition" (1.1.0)...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 2.14"
* provider.google: version = "~> 2.8"
* provider.hcloud: version = "~> 1.10"
* provider.ignition: version = "~> 1.1"
* provider.null: version = "~> 2.1"
* provider.random: version = "~> 2.1"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Рисунок 3.6 – Приклад виклику описаного у локальній директорії модулю

Таким чином було створено набір скриптів для автоматизації створення інфраструктури з метою автоматизації розгортання серверів. Описаний у додатку до дипломного проекту лістинг створює віртуальні машини у хмарних центрах обчислення даних Amazon, Google та Hetzner.

За необхідності лише перевірити, що Terraform планує виконати, слід використовувати команду “terraform plan”. Дана директива перевіряє

поточну конфігурацію ресурсів та виводить заплановані зміни без їх реального застосування. Такий підхід зручно використовувати у випадках, коли ведеться дослідження проблем або ж робота з важливими даними.

Кожен з модулів створює віртуальну машину заданої конфігурації ресурсів, що можуть бути змінені:

- Кількість ядер процесору;
- Об’єм оперативної пам’яті;
- Обсяг дискового простору.

За замовчуванням, створюються віртуальні машини з операційною системою CoreOS. Дана оперативна система створена на базі ядра Linux та вважається оптимальною у випадках коли головною метою використання віртуальних машин є запуск на них Docker контейнерів.

До особливостей даного дистрибутива можна віднести відсутність пакетного менеджеру. Операційна система містить встановлений Docker, отже є можливість завантажити будь-який контейнер що буде містити необхідні пакети та пакетні менеджери або ж оточення для запуску додатків створених за допомогою практично будь-яких технологій.

Кожна віртуальна машина створюється з заблокованими для доступу всіма портами, окрім 22-го. Даний порт є стандартним для демону SSH серверу. Даний протокол використовується для доступу до віртуальних машин для подальшого розгортання програмних додатків або інших дій. Віртуальні машини також є доступними за допомогою публічної IPv4 адреси. Проте, дані адреси є динамічними, що означає, що після перезапуску віртуальних машин, їх публічні IPv4 адреси змінюються на інші. Для забезпечення зручності доступу до віртуальних машин, генерується DNS запис типу A, що вказує на динамічну IPv4 адресу. Усі записи створюються у публічній DNS зоні vtatarin.space. Дана зона контролюється за допомогою сервісу AWS- Route53.

3.3 Конфігурування та розгортання тестів

Для автоматизації конфігурації віртуальних машин після їх створення за допомогою Terraform був використаний програмний комплекс Ansible. Даний додаток виконує команди через протокол віддаленого доступу до ресурсів SSH.

Бажана конфігурація віртуальних машин, котра має бути виконана Ansible міститься у файлах “плейбуках” у форматі YAML.

Це формат збереження даних, що вважається найбільш зручним для читання людиною. Концептуально цей формат близький до мов розмітки, проте вважається орієнтовним на зручність вводу-виводу типових структур даних для більшості мов програмування. Нині найбільш популярними зручними для читання людиною форматами зберігання даних вважаються саме YAML та JSON.

Серед конфігураційних файлів Ansible є наступні:

- Інвентар – файл, що містить список адрес всіх серверів, з якими планується взаємодія;
- Ролі – набір директорій, кожна з яких описує конфігурацію певного сервісу або стеку технологій;
- Змінні – набір змінних, що можуть бути використані для певних груп серверів;
- Списки дій – містять перелік ролей та груп серверів, які мають бути використані в процесі виконання описаних в ролях дій.

Для того, щоб запустити виконання описаних в ролях дій, необхідно виконати в командному рядку команду “ansible-playbook --python.yml --limit "aws" --diff”. Дана команда виконає список ролей, що є описаними у файлі python.yml, обравши серед них дії, що є поміченими тегом “aws”. Додатково буде виведена різниця між поточною конфігурацією серверу та тою, що була додана поточним виконанням команди.

За необхідністю лише перевірити, що поточна конфігурація Ansible планує виконати з сервером додатковий флаг “--check” може бути доданим. В такому випадку, заплановані зміни не будуть виконані з сервером, а лише виведуться заплановані зміни. Такий підхід є надзвичайно зручним при роботі з важливими реальними ресурсами.

Для того, щоб оцінити продуктивність кожного з розгорнутих серверів, було вирішено запустити навантажувальні тести на кожному з них. Для зручності розгортання тестових програмних додатків було вирішено використовувати Docker контейнер для кожного з додатків. Таким чином, обрані навантажувальні тести є крос-плафторменними. Технологія контейнеризації доступна для більшості операційних систем, тому тести можуть бути запущені навіть на звичайних персональних комп’ютерах.

Такий спосіб використання програмних додатків значно спрощує підбирання конфігурацій для різнотипних робочих оточень (зокрема, з різними операційними системами, наявними пакетами та ін.).

Створення власного контейнеру є простим. Головні команди, що мають бути виконані на етапі створення образу контейнеру описуються у файлі Dockerfile за допомогою спеціального синтаксису. Наприклад, для того, щоб створити контейнер з операційною системою Debian, додати туди середовище та OpenJDK і виконати команду для запуску тесту SuperPi після старту достатньо такого простого описання згаданих вище кроків у Dockerfile:

```
FROM openjdk

RUN yum update -y \
  && yum install -y python-pip wget\
  && pip install awscli

RUN wget https://github.com/Fewlaps/superpi/releases/download/v1.0.0/superpi-1.0.jar

RUN chmod +x superpi-1.0.jar

ADD test.sh /usr/bin/test.sh

ENV SERVICE="superpi" \
  SERVER_NAME="test" \
  S3_BUCKET="vtatarin-load-test" \
  SUPERPI_THREADS="1" \
  SUPERPI_ITER="32000000"

ENTRYPOINT []

CMD [ \
  "sh", \
  "-c", \
  "/usr/bin/test.sh" \
  ]
```

Рисунок 3.7 – Приклад Dockerfile

Для того, щоб створити образ даного контейнеру, необхідно виконати наступну команду: “docker build -t stress .”. Дана команда крок за кроком створить контейнер. Кожний крок є окремим шаром і при перевиконанні команди при відсутності змін у створених раніше шарах, шар не перезбирається, а використовується його кеш.

Обрана система контейнеризації використовує багатошарову файлову систему (Layer File System) при створенні кожного образу. Кожна команда з Dockerfile є окремим шаром, котрий доступний лише для читання після створення. Також ці шари інколи називаються проміжковими образами через те, що вони можуть бути використаними ми створенні образів інших контейнерів.

```
vtatarin@Valerils-MacBook-Pro ~/D/a/v/superpi> docker build -t stress .
Sending build context to Docker daemon   5.12kB
Step 1/8 : FROM openjdk
 ---> ef36deb98f03
Step 2/8 : RUN yum update -y   && yum install -y python-pip wget  && pip install awscli
 ---> Using cache
 ---> 4b0ddc60e329
Step 3/8 : RUN wget https://github.com/Fewlaps/superpi/releases/download/v1.0.0/superpi-1.0.jar
 ---> Using cache
 ---> 1b2eefa8cab8
Step 4/8 : RUN chmod +x superpi-1.0.jar
 ---> Using cache
 ---> f84fe33bd540
Step 5/8 : ADD test.sh /usr/bin/test.sh
 ---> df206398542d
Step 6/8 : ENV SERVICE="superpi"   SERVER_NAME="test"   S3_BUCKET="vtatarin-load-test"
 ---> Running in c7ab608bbfd9
Removing intermediate container c7ab608bbfd9
 ---> 683037343610
Step 7/8 : ENTRYPOINT []
 ---> Running in 62c9b66ae81d
Removing intermediate container 62c9b66ae81d
 ---> 20f1b3b0b79f
Step 8/8 : CMD [   "sh",   "-c",   "/usr/bin/test.sh"   ]
 ---> Running in 4daca43f1703
Removing intermediate container 4daca43f1703
 ---> f4a1b55417af
Successfully built f4a1b55417af
Successfully tagged stress:latest
```

Рисунок 3.8 – Приклад створення контейнеру з тестом SuperPI

Після створення контейнеру для того, щоб у майбутньому він міг бути завантажений на інші сервери, його можливо помістити у будь-яке з так званих регістрів контейнерів. Вони можуть бути як публічні (загальнодоступні) так і приватні (доступні з обмеженнями).

Одним з найбільш розповсюджених регістрів є Docker Hub. Отже, необхідно створити відповідний Dockerfile, його зібрати, а потім завантажити до публічного регістру контейнерів.

Для автоматизації описаного процесу у випадку простих у створенні контейнерів було використано мейкфайли. Це утиліта що автоматизує процес компонування та компіляції проектів за допомогою синтаксису спеціального виду. Найбільш широко використовується даний підхід при роботі з мовою програмування C. Приклад мейкфайлу для створення контейнеру з тестом SuperPI можна побачити на рисунку 3.9:

```
.PHONY: build login push

REGISTRY=bugaga1100/superpi
DOCKER_TAG=latest

build:
  docker build -t ${REGISTRY} .

login:
  docker login

push: build login
  docker push ${REGISTRY}:${DOCKER_TAG}
```

Рисунок 3.9 – Приклад мейкфайлу для створення контейнеру з тестом SuperPI

Структура проекту Ansible включає в себе наступні основні файли та директорії: директорія з інвентарем, конфігураційний файл ansible.cfg, набір скриптів для виконання roles та плейбук, що включає в себе комбінації інвентарних груп та ролей. Структура проекту зображена на рисунку 3.10:

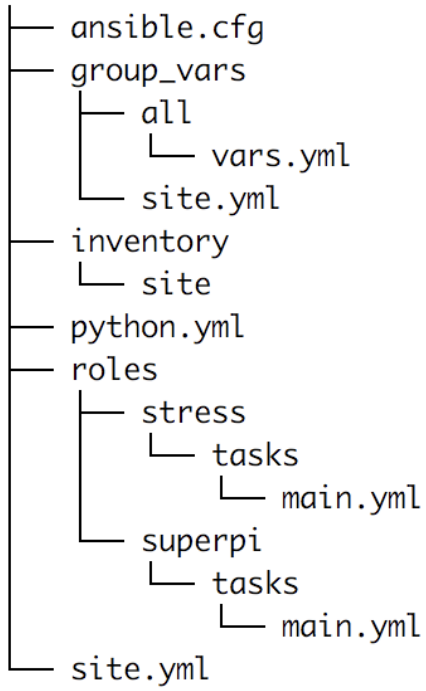

Рисунок 3.10 – Структура проекту Ansible

Основним файлом конфігурації сервісу Ansible є файл ansible.cfg. Він включає в себе набір головних директив, що визначають параметри роботи утиліти з керуючого серверу. Серед переліку конфігураційних директив є, наприклад, такі, що відповідають за конфігурацію клієнту SSH або ж визначають рівні логування сервісу. Також існує можливість задання не загальної конфігурації, а для певної групи серверів. Для виконання поставленого завдання необхідним був наступний файл конфігурації:

```
[defaults]
ansible_managed = Ansible managed: modified on %Y-%m-%d %H:%M by {uid} on {host}
inventory = inventory/
gathering = smart
fact_caching = jsonfile
fact_caching_connection = /tmp/.ansible_cache
fact_caching_timeout = 86400
remote_user = core
private_key_file = /Users/vtatarin/.ssh/vtatarin_cgn_ops
host_key_checking = False
```

Рисунок 3.11 – Приклад конфігураційного файлу Ansible

Зазначені вище директиви вказують на ім’я віддаленого користувача при встановленні SSH з'єднання, шлях до RSA ключа, який необхідно використовувати, налаштування HOST_KEY_CHECKING, що відповідає за включення/відімкнення перевірки ключа віддаленого серверу та інші.

Дані параметри визначені в блоці “defaults”, отже вони будуть обрані параметрами за замовчуванням для всіх серверів. Для того, щоб вказати специфічні параметри для певної групи серверів слід створити блок з такою ж назвою, яку назву носить група серверів. Наприклад, для того, щоб встановити параметри для роботи з серверами групи “aws” або “gcp” слід виконати наступні налаштування:

```
[aws]
ansible_managed = Ansible managed AWS changes: modified on %Y-%m-%d %H:%M by {uid} on {host}
inventory = inventory/aws
gathering = smart
fact_caching = jsonfile
fact_caching_connection = /tmp/.ansible_cache
fact_caching_timeout = 86400
remote_user = core
private_key_file = /Users/vtatarin/.ssh/gcp
host_key_checking = False

[gcp]
ansible_managed = Ansible managed GCP changes: modified on %Y-%m-%d %H:%M by {uid} on {host}
inventory = inventory/gcp
gathering = smart
fact_caching = jsonfile
fact_caching_connection = /tmp/.ansible_cache
fact_caching_timeout = 86400
remote_user = core
private_key_file = /Users/vtatarin/.ssh/aws
host_key_checking = False
```

Рисунок 3.12 – Конфігураційні параметри для груп серверів “aws” та “gcp”

Серед конфігураційних файлів Ansible є файл адресами всіх серверів, на яких мають бути виконані віддалені команди. Даний файл називається інвентарним та знаходиться у кореневій директорії проекту з Ansible. Він містить рядки такого вигляду:

```
[aws]
# 2vCPU/4GB
t3-medium.vtatarin.space
c4-large.vtatarin.space
t2-medium.vtatarin.space
t3-medium.vtatarin.space
t3a-medium.vtatarin.space

[hetzner]
cx21.vtatarin.space
cx31.vtatarin.space


[gcp]
n1-highcpu-2.vtatarin.space
n1-standard-2.vtatarin.space
```

Рисунок 3.13 – Приклад інвентарного файлу для Ansible

Ansible для виконання певних команд використовує набір команд описаних у YAML. Такі файли носять назву плейбуки. Для запуску

контейнерів на віддалених серверах стандартний плейбук має наступний вигляд:

```
---
- name: Run Stress container
  docker_container:
    name: stress-{{ ansible_date_time.hour }}-{{ ansible_date_time.minute }}
    image: bugaga1100/stress
    state: started
    env:
      AWS_ACCESS_KEY_ID: AKIAYYZPATU21232UMFPH
      AWS_SECRET_ACCESS_KEY: xihwkg2tWO21DC/ksady5vim1WQBbYcLgh+9GCyvj
      AWS_DEFAULT_REGION: us-east-1
      SERVER_NAME: "{{ inventory_hostname.split(\".\")[0] }}"
      CPU_CORES: "{{ stress_cores }}"
      TIMEOUT: "{{ stress_timeout }}m"

- pause:
    minutes: "{{ stress_timeout }}"
```

Рисунок 3.14 – Приклад плейбуку Ansible

3.4 Інструкція користувачу

Для того, щоб розпочати роботу з розробленою системою необхідно дотримуватися наступного переліку дій:

1. Обрати бажаний центр обчислення даних серед тих, що підтримуються (AWS, GCP, Hetzner);
2. Обрати типи віртуальних машин, що мають бути розгорнуті, додати їх до конфігурації Terraform (рис. 3.15);

```
module "aws" {
  source = "../modules/aws"

  service = "${local.service}"
  type    = "on-demand"

  instance_type = [
    "t2.medium",
    "t2.large",
  ]
}
```

Рисунок 3.15 – Конфігурація Terraform для AWS

3. Створити віртуальні машини обраного типу за допомогою Terraform (рис. 3.16);

```
vtatarin@Valeriis-MacBook-Pro ~/D/d/v/test> terraform apply --target=module.aws

Apply complete! Resources: 6 added, 0 changed, 0 destroyed.

Outputs:

aws_dns = [
    t2-medium.vtatarin.space,
    t2-large.vtatarin.space
]
```

Рисунок 3.16 – Створення віртуальних машин у AWS

4. Адреси створених у п. 3 серверів з блоку Outputs необхідно перенести до інвентарного файлу Ansible (рис. 3.17);

```
[aws]
# 2vCPU/4GB
t2-medium.vtatarin.space
t2-large.vtatarin.space
```

Рисунок 3.17 – Інвентарний файл Ansible

5. Запустити у консолі скрипт aggregator.sh, що виконає конфігурацію серверів, запустить навантажувальні тести на них та проведе агрегацію результатів тестування у файли формату CSV (рис. 3.18);

```
vtatarin@Valeriis-MacBook-Pro ~/P/diploma> bash -c aggregator.sh
[2019-06-11 03:38:45] INFO: Working with /raw-superpi/t2-medium...
[2019-06-11 03:38:45] INFO: Working with /raw-superpi/t2-large...
[2019-06-11 03:38:45] INFO: Success!
```

Рисунок 3.18 – Приклад запуску aggregator.sh

6. Результати тестів будуть будуть збережені у файлах stress.csv та superpi.csv.

3.5 Методика тестування

Серед заявлених характеристик віртуальних машин у переліку параметрів завжди можна побачити наступні параметри:

- Об’єм оперативної пам’яті (RAM);
- Кількість ядер центрального процесору (CPU);
- Пропускна здатність мережевого інтерфейсу;
- Максимальний об’єм дискового простору, тип дисків та їх продуктивність (IOPS).

Більшість перелічених параметрів можна просто порівняти між собою для різних типів віртуальних машин та для різних ЦОД. Проте, щодо центрального процесору, то завжди зазначається лише кількість віртуальних (або ж реальних) ядер. Також інколи зазначають аналог за продуктивністю серед реальних існуючих процесорів. Так, наприклад, AWS для свого сервісу EC2 заявляюсь, що кожне віртуальне ядро є віртуальним потоком процесору Intel модельного ряду Xeon, окрім типу EC2 T2, але не вказується, якого саме Intel Xeon.

Саме тому аналіз продуктивності саме центрального процесору віртуальних машин є найбільш важливим.

Для проведення тестування будуть проведені тести SuperPI з кількістю ітерацій у 32000000 у однопотоковому режимі та stress-ng у багатопотоковому. Кількість потоків для виконання обчислень дорівнює зазначеному провайдером послуг числу віртуальних ядер наданої віртуальної машини.

Використані тести запускатимуться саме у режимі стресового навантаження для центрального процесору, тому кількість оперативної пам'яті наявної у віртуальної машини не відіграє значущої ролі у результатаї тестувань.

Через зазначені особливості використаного ПЗ для проведення дослідження, всі обрані віртуальні машини були поділені на 4 групи, за кількістю наявних віртуальниї ядер.

Зазвичай у більшості провайдерів хмарних послуг наявні віртуальні машини з 2, 4, 8 та 16 віртуальними ядрами. Конфігурації з більшою та меншою кількістю ядер є менш розповсюдженими та менше використовуються, адже зазвичай від кількості ядер залежить й кількість доступної оперативної пам’яті.

Таблиця 3.1 – Група віртуальних машин з двома ядрами

Провайдер	**Тип віртуальних машин**
AWS	c4.large, c5.large, m4.large, m5.large, m5a.large, m4.large, r5.large, t2.large, t2.medium, t3.large, t3.medium, t3a.medium
GCP	n1-highcpu-2, n1-highmem-2, n1-standard-2
Hetzner	cx21, cx31

Таблиця 3.2 – Група віртуальних машин з чотирма ядрами

Провайдер	**Тип віртуальних машин**
AWS	c4.xlarge, c5.xlarge, m4.xlarge, m5.xlarge, m5a.xlarge, r4.xlarge, r5.xlarge, r5a.xlarge, t2.xlarge, t3.xlarge, t3a.xlarge
GCP	n1-highcpu-4, n1-highmem-4 n1-standard-4
Hetzner	cx41

Таблиця 3.3 – Група віртуальних машин з восьма ядрами

Провайдер	**Тип віртуальних машин**
AWS	c4.2xlarge, c5.2xlarge, m4.2xlarge, m5.2xlarge, m5a.2xlarge, r4.2xlarge, r5.2xlarge, t2.2xlarge, t3.2xlarge, t3a.2xlarge
GCP	n1-highcpu-8, n1-highmem-8, n1-standard-8
Hetzner	cx51

Таблиця 3.4 – Група віртуальних машин з шістнадцятьма ядрами

Провайдер	**Тип віртуальних машин**
AWS	c4.4xlarge, c5.4xlarge, m4.4xlarge, m5.4xlarge, m5a.4xlarge, r4.4xlarge,
GCP	n1-highcpu-16, n1-highmem-16, n1-standard-16
Hetzner	–

Вище наведені групи віртуальних машин (таблиці 3.1, 3.2, 3.3, 3.4) для кожного з тестованих провайдерів. Тести будуть проведені для порівняння віртуальних машин саме у таких групах.

Як можна побачити, кожен з ЦОД надає віртуальні машини певних класів, наприклад, серед протестованих машин AWS EC2 є такі типи:

- T – віртуальні машини загального призначення з наявністю короткочасного підвищення продуктивності, найдешевший тип віртуальних машин;
- M – віртуальні машини загального призначення з найкращим балансом кількості ядер/оперативної пам’яті. Мають підвищену

пропускну здатність мережевих інтерфейсів, у порівнянні з T типом;

- R – віртуальні машини з підвищеним об’ємом оперативної пам’яті. Даний тип віртуальних машин слід використовувати для розгортання ПЗ з підвищеними вимогами до об’єму RAM;
- C – віртуальні машини, оптимізовані для обчислювальних операцій. Вважається, що віртуальні ядра саме цих машин є найбільш продуктивними серед усіх з точки зору CPU оопераций.

Сервіс AWS EC2 є найстарішим сервісом надання віртуальних машин серед оглянутих. За час існування EC2, була зроблена велика кількість покращень, тому існують покоління практично кожного з зазначених вище типів.

Найбільш актуальними є такі покоління віртуальних машин – T3, M5, R4, C5. Вони створені за допомогою віртуалізації нового типу – AWS Nitro. Також ці віртуальні машини підтримують NVMe SSD для зберігання даних, що є одним з найшвидших типів non-volatile пам’яті.

GCP має менший вибір серед типів віртуальних машин. Проте, лише цей провайдер дає змогу створювати власні типи віртуальних машин з власноруч налаштованими параметрами. Серед стандартних наявних віртуальних машин є наступні:

- standard – віртуальні машини загального призначення, мають найбільш рівномірно збалансоване відношення кількості обчислювальних віртуальних ядер до об’єму оперативної пам’яті;
- highcpu – оптимізовані для обчислювальних операцій, виконуваних за допомогою центрального процесору, віртуальні машини;
- highmem – віртуальні машини з підвищеною кількістю оперативної пам’яті.

Hetzner Cloud має найменший вибір серед типів машин. Вони надають можливість створення лише віртуальних машин, або ж оренда реальних виділених серверів. Іменування віртуальних машин загального типу – CX.

Зважаючи на оголошення GCP, AWS та Hetzner, слід зауважити, що віртуальними хостами для своїх сервісів надання віртуальних машин провайдери використовують сервери на базі процесорів Intel. Лише AWS представили нове покоління віртуальних машин, що організовані на AMD хостах. Це віртуальні машини типів. M5a та T3a. Дані типи віртуальних машин є дешевшими та з дещо нижчою продуктивністю, а от на скільки нижчою – покажуть результати тестування.

3.5 Результати тестування

Результати тестування групи двоядерних віртульних машин під навантаженням Stress-ng наведені на рисунку 3.19.

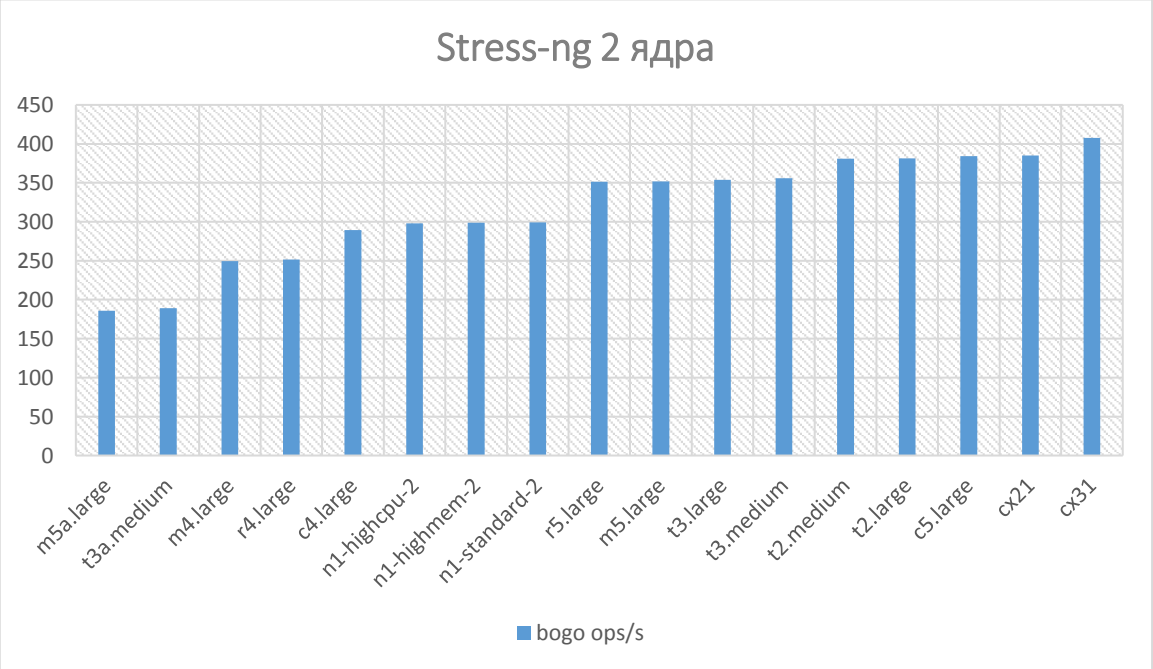

Рисунок 3.19 – Результати Stress-ng для двоядерних віртуальних машин (більше – краще)

Результати тестування групи двоядерних віртульних машин під навантаженням SuperPI наведені на рисунку 3.20.

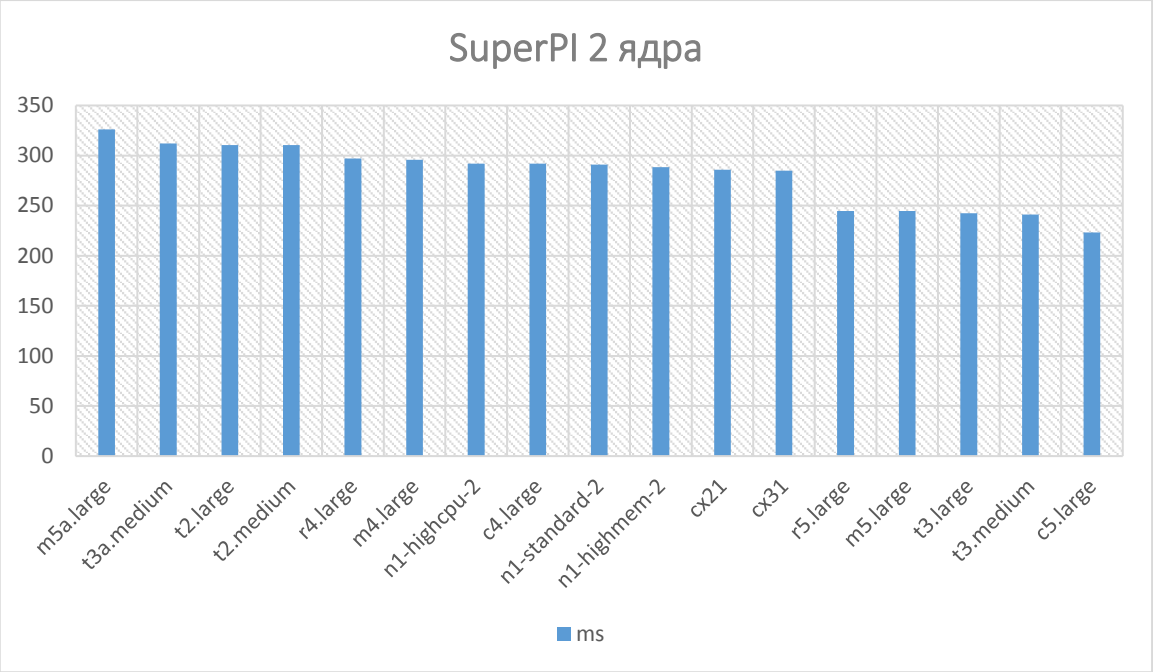

Рисунок 3.20 – Результати SuperPI для двоядерних віртуальних машин (менше – краще)

Серед протестованих двоядерних віртуальних машин найповільнішими є віртуальні машини T3a/M5a – вони базуються на хостах з процесорами AMD. Найкращу продуктивність у багатопоточному режимі показують віртуальні машини від Hetzner та AWS сімейства C5. Подібна тенденція є правдивою і для решти віртуальних машин різних масштабів.

Результати тестування групи чотириядерних віртульних машин під навантаженням Stress наведені на рисунку 3.21.

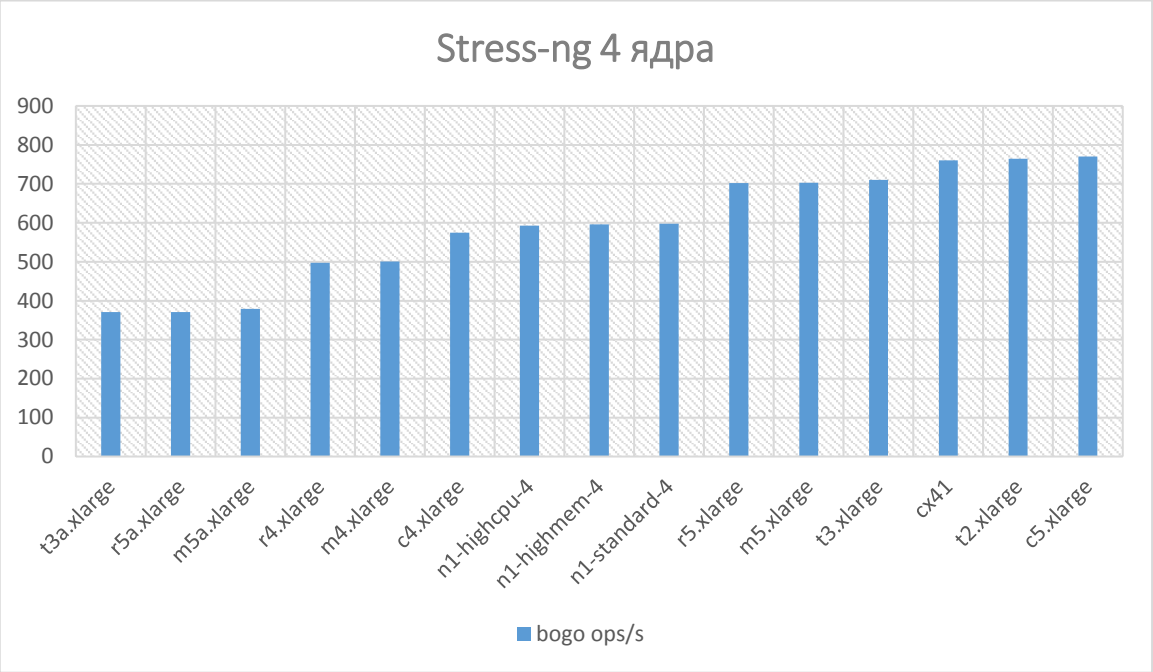

Рисунок 3.21 – Результати Stress-ng для чотириядерних віртуальних машин (більше – краще)

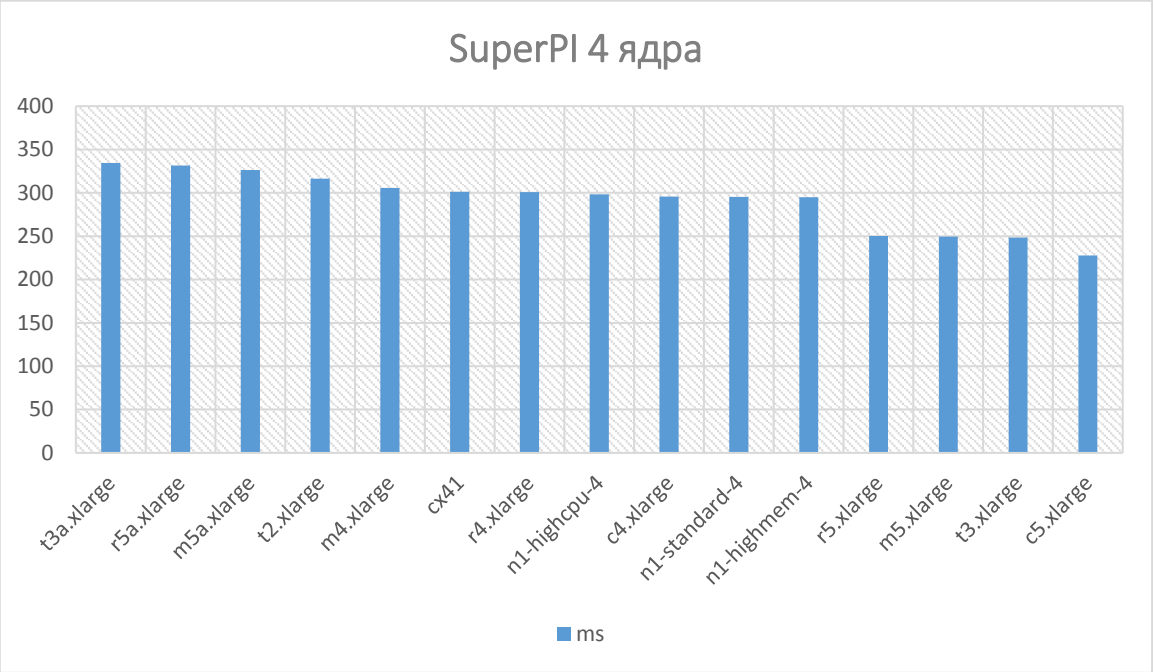

Рисунок 3.22 – Результати SuperPI для чотириядерних віртуальних машин (менше – краще)

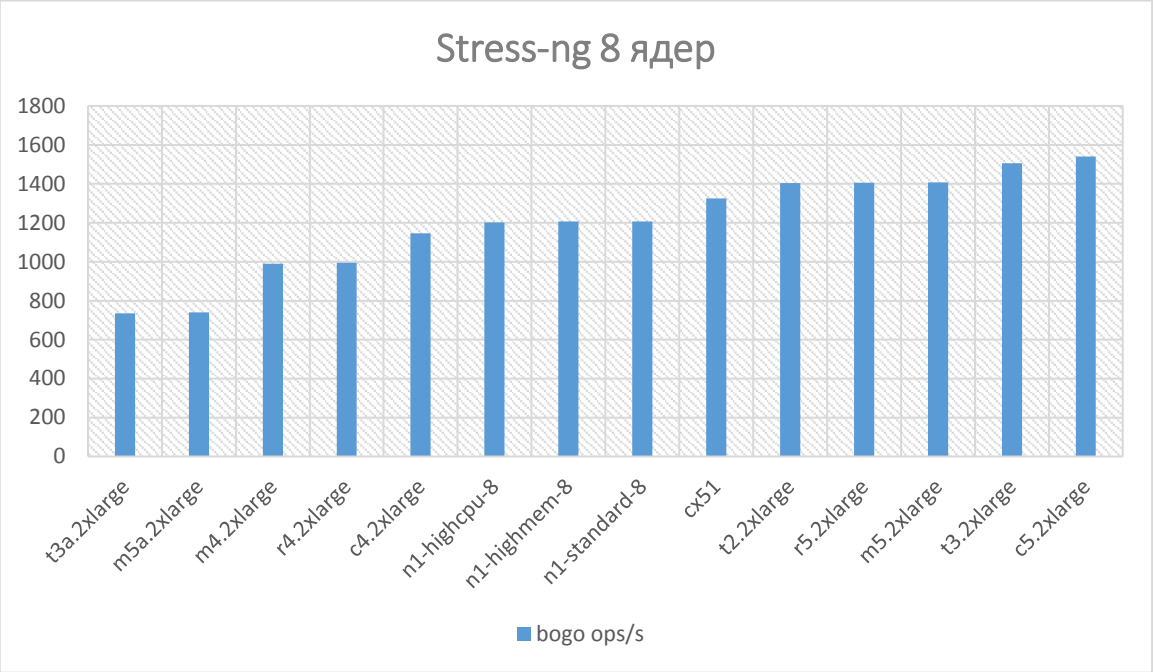

Рисунок 3.23 – Результати Stress-ng для восьмиядерних віртуальних машин (більше – краще)

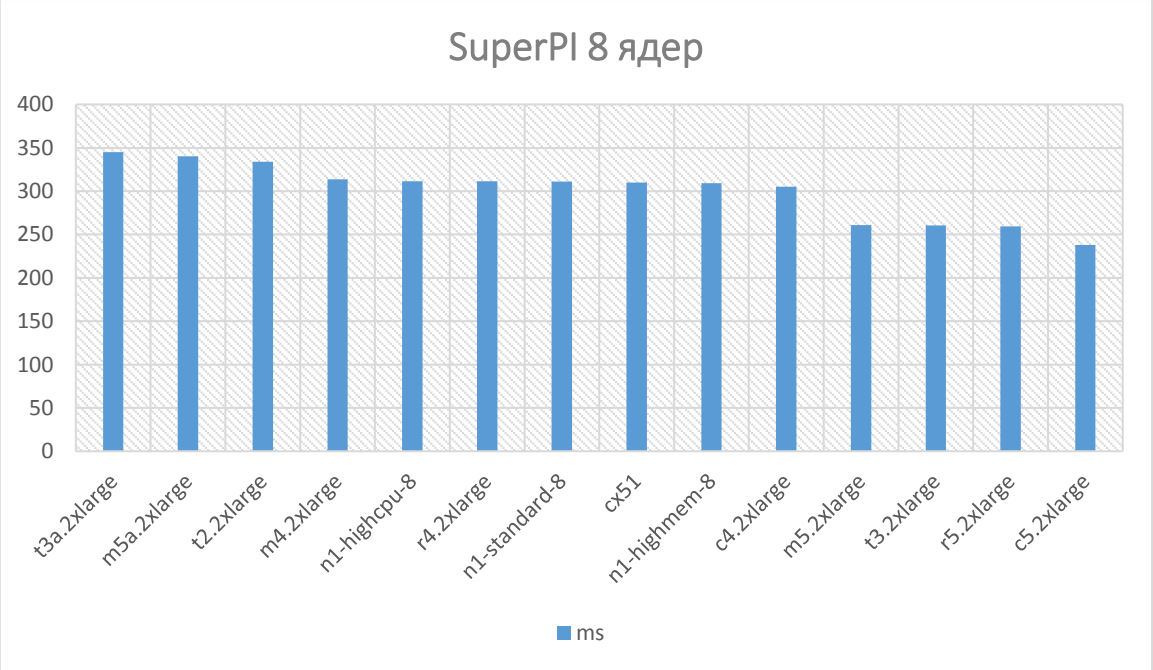

Рисунок 3.24 – Результати SuperPI для восьмиядерних віртуальних машин (менше – краще)

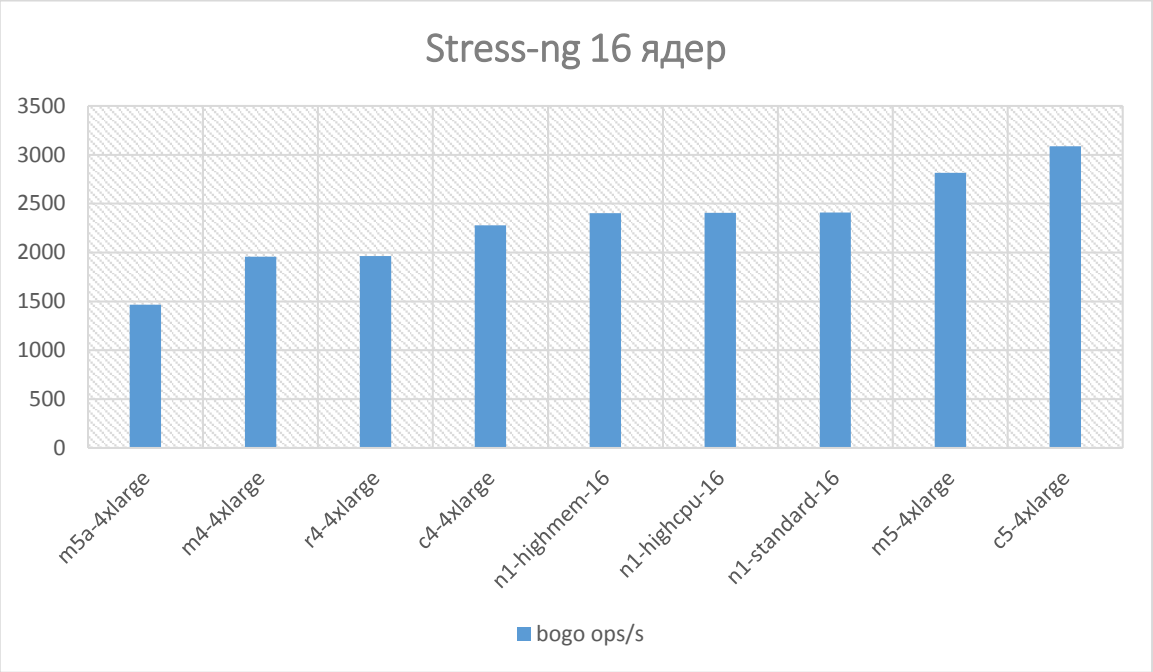

Рисунок 3.25 – Результати Stress-ng для шістнадцятиядерних віртуальних машин (більше – краще)

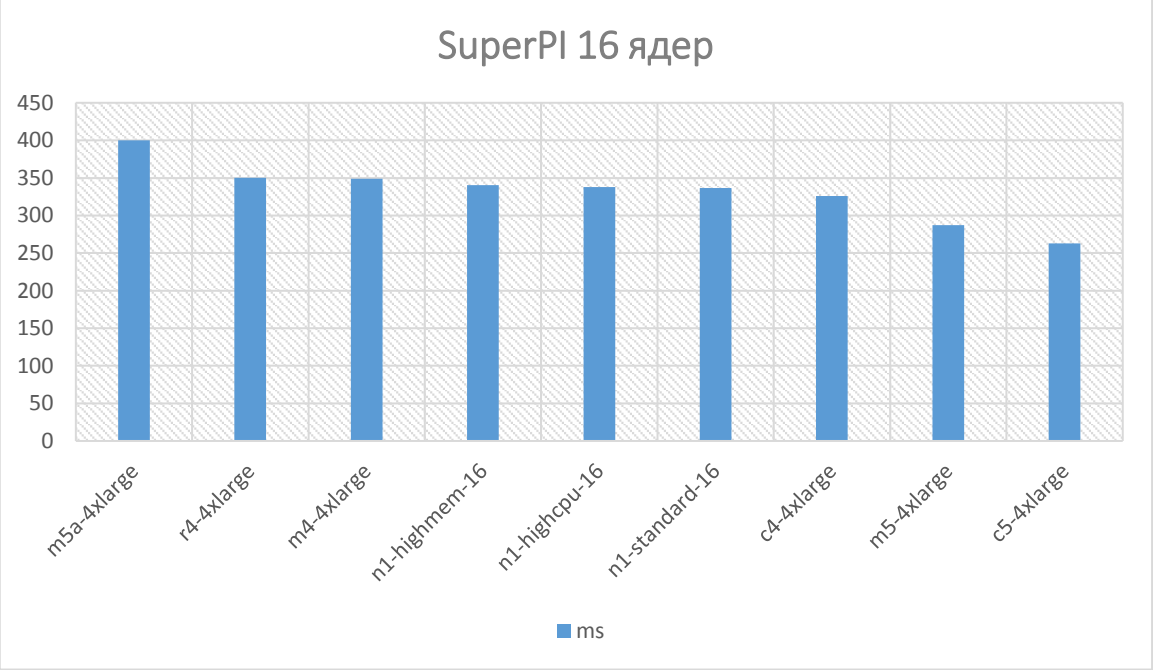

Рисунок 3.26 – Результати SuperPI для шістнадцятиядерних віртуальних машин (менше – краще)

З отриманих результатів тестувань видно, що серед усіх груп віртуальних машин набільш продуктивними у більшості тестів є

віртуальні машини від AWS типу C5, у середньому на 10% швидші за найближчого конкурента у кожній категорії. Не набагато менш продуктивними є віртуальні машини від AWS групи M5 та R5.

Віртуальні машини від GCP у середньому є на 10-14 % повільніші за своїх конкурентів від AWS.

Обчислювальні потужності від Hezner є лідерами досліджень у класі двоядерних віртуальних машин та показують зовсім невелике відставання від аналогів AWS у решті тестів.

Віртуальні машини від AWS типу T виконували обчислення у режимі підвищеної продуктивності. Використання даного типу віртуальних машин є оправданим у випадках, коли навантаження на систему відбувається нерівномірно з плином часу.

Аутсайдерами всіх тестів є віртуальні машини типів M5a та T3a.

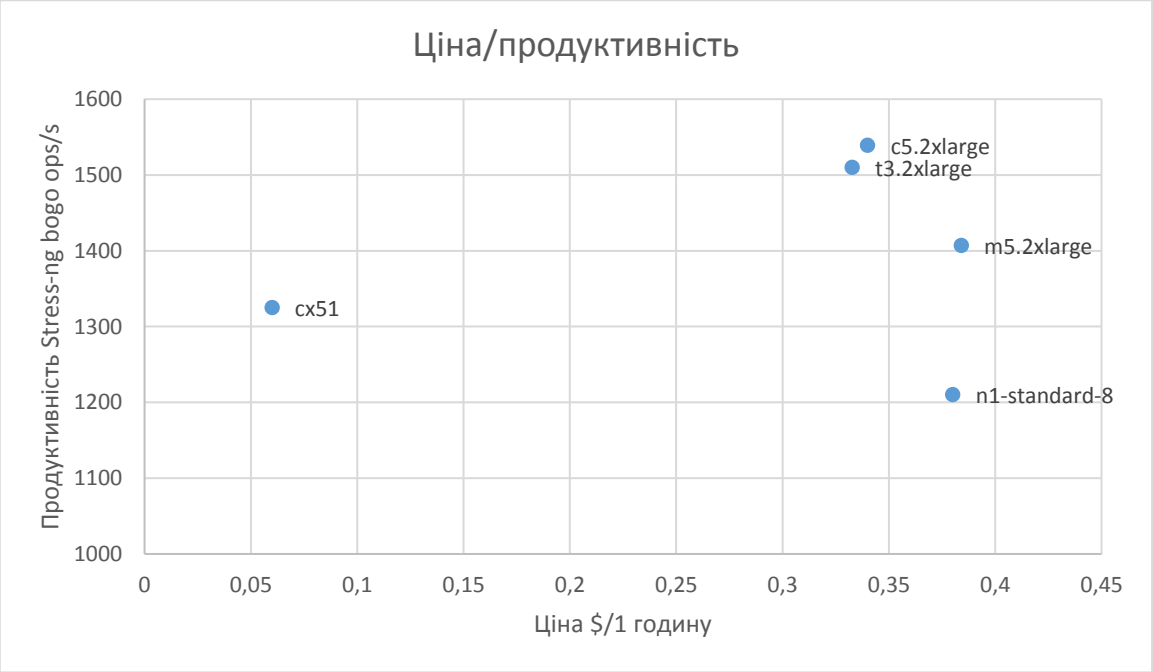

Рисунок 3.27 – Співвідношення ціни та продуктивності для восьмиядерних віртуальних машин (лівий верх- накраще)

З рисунку 3.27 видно загальну динаміку найкращих типів віртуальних машин. Серед лідерів зі схожими параметрами

наневигідніша пропозиція – від Google Cloud Platform. А от найкраще себе проявляє з точки зору бюджету – Hetzner Cloud. Їх віртуальні машини є надзвичайно дешевими при невеликих витратах. Упевненими суперниками будь-яким хмарним PaaS провайдерів є віртуальні машини від AWS типів T3, C5, M5.

Необхідно також пам’ятати, що дослідження були проведені за використанням синтетичних стрес тестів, що виконували математичні обчислення, операції архівування та розархівування або ж шифрування та дешифрування даних. Для отримання більш точних результатів тестування для конкретних випадків використання, необхідно навантажувати систему власним ПЗ. Розроблена система є модульною, тому з легкістю дозволить це зробити.

3.6 Висновки до розділу 3

У даному розділі було описане проектування системи автоматизації розгортання та аналізу продуктивності серверів, її розробка та принципи роботи. Були використані досліджені інструменти для розробки системи, а саме Terraform та Ansible для конфігурації та розгортання серверів та SuperPI, Stress-ng для проведення стрес тестів.

На основі отриманих даних можна зробити висновок, що не слід використовувати віртуальні машини від AWS типу M5a та T3a, оскільки вони значно повільніші за свої аналоги, особливо при використанні багатоядерних віртуальних машин, хоча ціна нижча лише на 10% за свої аналоги.

Віртуальні машини від GCP є практично однаковими за продуктивністю центрального процесору і відрізняються лише об’ємом оперативної пам’яті. Це досить логічно, оскільки ціна за кожну

віртуальну машину у GCP формується як сума ціни за кожне віртуальне ядро процесору та за кожний гігабайт оперативної пам’яті.

Також була розроблена детальна інструкція користувачеві, котра повністю пояснює принципи взаємодії з системою. Користувач з легкістю зможе оперувати системою при детальному вивченні інструкції при виході нових типів віртуальних машин у провайдерів хмарних центрів обчислення даних, що підтримуються системою на момент розробки.

Залежність продуктивності віртуальних машин від використаного при їх створенні гіпервізору не є суттєвою. Розглянуті типи серверів від GCP, Hetzner та частина AWS були створені за домогою KVM гіпервізору, а от більшість у AWS з використанням Xen. Така відсутність відмінностей можлива через викоористання різних типів та поколінь процесорів для хост машин. Продуктивність на високому рівні віртуальних машин типу AWS T3 та M5 також може бути пов'язана з використанням технології Nitro, що у поєднанні з сучасними поколіннями процесорів хост машин робить згадані класи віртуальних машин одними з найкращих на ринку.

ВИСНОВКИ

Існуючих аналогів розробленій системі виявлено досить мало. Всі вони не є відкритими системами для проведення аналізу продуктивності, а лише публікують результати проведення досліджень. Опубліковані у вільному доступі результати не є детальними та досить часто не містять актуальної інформації.

Було обґрунтовано вибір технологій, що будуть використовуватися для системи. Серед них є такі актуальні технології автоматизації роозгортання серверів як Terraform, Ansible. Були обрані досить широко використовувані стрес тести Stress-ng та SuperPI та найпопулярніша технологія розгортання ПЗ – Docker.

Було проведено проектування системи, описано методику розгортання та аналізу продуктивності серверів, створено інструкцію користувача.

Розроблена система повністю задовольняє поставлені вимоги. Вона дозволяє проводити розгортання та аналіз продуктивності серверів створених у декількох хмарних центрах обробки даних. Через реалізовану модульну архітектуру можливо з легкістю додавати функціонал для підтримки API інших платформ. Система проводить конфігурацію віртуальних машин, створюючи максимально універсальну робочу середу для будь-яких потреб. Розроблена методика тестування є універсальною та може бути використана для будь-яких віртуальних машин.

Отримані за допомогою розрообленої системи результати тестування віртуальних машин можуть бути використані при проектуванні інфраструктур реальних проектів, що може підвищити швидкодію та зменшити інфраструктурні витрати.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cloud Computing Forecasts and Market Estimates 2018 [Електронний ресурс]: Режим доступа: https://www.forbes.com/sites/louiscolumbus/2018/09/23/roundup-of-cloud-computing-forecasts-and-market-estimates-2018/#5fdbc645507b

2. Cloud Computing [Електронний ресурс]: Режим доступа: https://globussoft.com/cloud-computing/

3. Cloud Computing expectations [Електронний ресурс]: Режим доступа: https://www.ey.com/in/en/industries/technology/cloud-computing-adoption-in-india--expectations-from-cloud-computing-vendors

4. What is Cloud Computing – Everything you need to know [Електронний ресурс]: Режим доступа: https://www.zdnet.com/article/what-is-cloud-computing-everything-you-need-to-know-from-public-and-private-cloud-to-software-as-a/

5. Top 5 Cloud Vendors [Електронний ресурс]: Режим доступа: https://cloudwars.co/worlds-top-5-cloud-vendors-cloud-wars/

6. Infrastructure as a Service [Електронний ресурс]: Режим доступа: https://en.wikipedia.org/wiki/Infrastructure_as_a_service

7. Platform as a Service [Електроонний ресурс]: Режим доступа: https://en.wikipedia.org/wiki/Platform_as_a_service

8. SRE fundamentials: SLIs, SLAs, SLOs [Електронний ресурс]: Режим доступа: https://cloud.google.com/blog/products/gcp/sre-fundamentals-slis-slas-and-slos

9. Software as a Service [Електронний ресурс]: Режим доступа: https://en.wikipedia.org/wiki/Software_as_a_service

10. Hypervisor [Електронний ресурс]: Режим доступа: https://en.wikipedia.org/wiki/Hypervisor

11. VMware ESXi [Електронний ресурс]: Режим доступа: https://en.wikipedia.org/wiki/VMware_ESXi

12. Binary translation and related topics [Електронний ресурс]: Режим доступа: https://www.sciencedirect.com/topics/computer-science/binary-translation

13. Introduction to KVM [Електронний ресурс]: Режим доступа: https://mkdev.me/en/posts/virtualization-basics-and-an-introduction-to-kvm

14. Introduction to Xen [Електронний ресурс]: Режим доступа: https://doc.opensuse.org/documentation/leap/virtualization/html/book.virt/cha.xen.basics.html

15. AWS Nitro [Електронний ресурс]: Режим доступа: https://www.metricly.com/aws-nitro/

16. Docker introduction [Електронний ресурс]: Режим доступа: https://docs.docker.com/engine/docker-overview/

17. Docker adoption statistics [Електронний ресурс]: Режим доступа: https://www.datadoghq.com/docker-adoption/

18. Container orchestration leaders [Електронний ресурс]: Режим доступа: https://www.g2.com/categories/container-orchestration

19. Hashicorp Configuration Language [Електронний ресурс]: Режим доступа: https://github.com/hashicorp/hcl

20. DevOps fundamentials: Infrastructure as Code [Електронний ресурс]: Режим доступа: https://puppet.com/solutions/infrastructure-as-code

21. RedHat Ansible management [Електронний ресурс]: Режим доступа: https://www.redhat.com/en/technologies/management/ansible

22. Stress-ng [Електронний ресурс]: Режим доступа: https://wiki.ubuntu.com/Kernel/Reference/stress-ng

23. HWBot SuperPI leaderboard [Електронний ресурс]: Режим доступа: https://hwbot.org/benchmark/superpi_-_1m/

ДОДАТОК А

Код Terraform модулю aws

```
data "aws_ami" "experiment" {
  most_recent = true

  filter {
    name   = "name"
    values = ["${lookup(var.ami_name, var.ami_distrib)}*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  filter {
    name   = "architecture"
    values = ["x86_64"]
  }

  owners = ["${lookup(var.ami_owner, var.ami_distrib)}"]
}

resource "aws_instance" "experiment" {
  count         = "${length(var.instance_type) * "${var.type == "spot" ? 0 :
1}"}"
  ami           = "${data.aws_ami.experiment.id}"
  instance_type = "${var.instance_type[count.index]}"
```

```
  key_name = "${var.admin_key_name}"

  user_data = "${data.ignition_config.default.*.rendered[count.index]}"

  vpc_security_group_ids = [
    "${aws_security_group.default.id}",
  ]

  subnet_id                   =
"${element(random_shuffle.subnet.*.result[count.index], 0)}"
  associate_public_ip_address = true

  tags {
    Name    = "${var.service}-${var.instance_type[count.index]}"
    Service = "${var.service}"
    Spot    = "false"
  }

  root_block_device {
    volume_type           = "${var.root_volume_type}"
    volume_size           = "${var.root_volume_size}"
    delete_on_termination = true
  }
}
data "ignition_file" "hostname" {
  count      = "${length(var.instance_type)}"
  path       = "/etc/hostname"
  filesystem = "root"
```

```
  mode       = "420"

  content {
    content = <<EOF
${replace("${var.instance_type[count.index]}", ".", "-")}
EOF
  }
}

data "ignition_config" "default" {
  count = "${length(var.instance_type)}"

  files = [
    "${data.ignition_file.hostname.*.id[count.index]}",
  ]
}
output "dns" {
  value = [
    "${aws_route53_record.experiment.*.fqdn}",
  ]
}
data "aws_route53_zone" "main" {
  name = "${var.r53_public_zone}."
}

# Route53 records
resource "aws_route53_record" "spot_experiment" {
  count   = "${length(var.instance_type) * "${var.type == "spot" ? 1 :
0}"}"
```

```
  zone_id = "${data.aws_route53_zone.main.zone_id}"

  name = "spot-${replace("${var.instance_type[count.index]}", ".", "-
")}"
  type = "A"
  ttl  = "300"

  records = [

"${aws_spot_instance_request.spot_experiment.*.public_ip[count.index]
}",
  ]
}

resource "aws_route53_record" "experiment" {
  count   = "${length(var.instance_type) * "${var.type == "spot" ? 0 :
1}"}"
  zone_id = "${data.aws_route53_zone.main.zone_id}"

  name = "${replace("${var.instance_type[count.index]}", ".", "-")}"
  type = "A"
  ttl  = "300"

  records = [
    "${aws_instance.experiment.*.public_ip[count.index]}",
  ]
}
# Default security group and rules
resource "aws_security_group" "default" {
```

```
  name        = "${var.service}-default"
  description = "Default security group for ${var.service}"
  vpc_id      = "${var.vpc_id}"

  tags {
    Name = "${var.service}-default"
  }
}

resource "aws_security_group_rule" "default_self_egress" {
  security_group_id = "${aws_security_group.default.id}"
  self              = true

  type      = "egress"
  from_port = "0"
  to_port   = "0"
  protocol  = "-1"
}

resource "aws_security_group_rule" "default_public_egress" {
  security_group_id = "${aws_security_group.default.id}"
  cidr_blocks       = ["0.0.0.0/0"]

  type      = "egress"
  from_port = "0"
  to_port   = "0"
  protocol  = "-1"
}
```

```
resource "aws_security_group_rule" "default_self_ingress_icmp" {
  security_group_id = "${aws_security_group.default.id}"
  self              = true

  type      = "ingress"
  from_port = -1
  to_port   = -1
  protocol  = "icmp"
}

resource "aws_security_group_rule" "default_public_ingress_ssh" {
  security_group_id = "${aws_security_group.default.id}"
  cidr_blocks       = ["0.0.0.0/0"]

  type      = "ingress"
  from_port = 22
  to_port   = 22
  protocol  = "tcp"
}
data "aws_vpc" "experiment" {
  id = "${var.vpc_id}"
}

data "aws_subnet_ids" "experiment" {
  vpc_id = "${data.aws_vpc.experiment.id}"
}

resource "random_shuffle" "subnet" {
  count = "${length(var.instance_type)}"
```

```
  input = ["${data.aws_subnet_ids.experiment.ids}"]

  result_count = 1
}
```

Код Terraform модулю gcp

```
data "google_compute_image" "os" {
  family  = "${lookup(var.os_family, var.os_distrib)}"
  project = "${lookup(var.os_project, var.os_distrib)}"
}

resource "google_compute_instance" "default" {
  count = "${length(var.instance_type)}"
  name  = "${var.service}-${var.instance_type[count.index]}"

  # https://cloud.google.com/dataproc/docs/concepts/compute/custom-machine-types
  machine_type = "${var.instance_type[count.index]}"
  zone         = "${data.google_client_config.current.region}-b"

  description = "Instance for ${var.service} purposes"
  hostname    = "${var.instance_type[count.index]}.${var.r53_public_zone}"

  boot_disk {
    auto_delete = true

    initialize_params {
      type  = "${var.boot_volume_type}"
```

```
    size  = "${var.boot_volume_size}"
    image = "${data.google_compute_image.os.self_link}"
   }
  }

  network_interface {
   network = "default"

   access_config {
    # Ephemeral IP
   }
  }

  metadata {
   ssh-keys = "${var.os_distrib}:${file(var.ssh_pub_key_file)}"
  }

  tags = [
   "${var.service}",
  ]

  labels {
   service = "${var.service}"
  }
 }
 output "dns" {
  value = [
   "${aws_route53_record.experiment.*.fqdn}",
  ]
```

```
}
data "aws_route53_zone" "main" {
  name = "${var.r53_public_zone}."
}

# Route53 records
resource "aws_route53_record" "experiment" {
  count   = "${length(var.instance_type)}"
  zone_id = "${data.aws_route53_zone.main.zone_id}"

  name = "${var.instance_type[count.index]}"
  type = "A"
  ttl  = "300"

  records = [

"${google_compute_instance.default.*.network_interface.0.access_confi
g.0.nat_ip[count.index]}",
  ]
}
data "google_client_config" "current" {}

variable "service" {}

variable "os_distrib" {
  default = "core"
}

variable "os_family" {
```

```
  default = {
    core   = "coreos-stable"
    centos = "centos-7"
  }
}

variable "os_project" {
  default = {
    core   = "coreos-cloud"
    centos = "gce-uefi-images"
  }
}

variable "ssh_pub_key_file" {
  default = "/Users/vtatarin/.ssh/vtatarin_cgn_ops.pub"
}

variable "instance_type" {
  type = "list"
}

variable "boot_volume_type" {
  default = "pd-ssd"
}

variable "boot_volume_size" {
  default = "100"
}
```

```
variable "r53_public_zone" {
  default = "vtatarin.space"
}

variable "r53_public_zone_id" {
  default = ""
}
```

Код Terraform модулю hetzner

```
data "hcloud_ssh_key" "admin_key" {
  fingerprint = "${var.admin_key_fp}"
}
data "ignition_user" "core" {
  name                = "core"
  ssh_authorized_keys = ["${file("${var.ssh_key_file}.pub")}"]
}

data "ignition_config" "default" {
  users = [
    "${data.ignition_user.core.id}",
  ]

  systemd = [
    "${data.ignition_systemd_unit.docker_service.id}",
    "${data.ignition_systemd_unit.docker_volume.id}",
  ]

  filesystems = [
    "${data.ignition_filesystem.docker.id}",
```

```
  ]
}

data "ignition_filesystem" "docker" {
  name = "docker"

  mount {
    device          = "/dev/sdb"
    format          = "xfs"
    wipe_filesystem = true
  }
}

data "ignition_systemd_unit" "docker_service" {
  name = "docker.service"

  dropin {
    name = "10-wait-docker.conf"

    content = <<EOF
[Unit]
After=local-fs.target
Requires=local-fs.target
EOF
  }
}

data "ignition_systemd_unit" "docker_volume" {
  name = "var-lib-docker.mount"
```

```
  content = <<EOF
[Unit]
Before=local-fs.target

[Mount]
What=/dev/sdb
Where=/var/lib/docker
Type=xfs

[Install]
WantedBy=local-fs.target
EOF
}
data "hcloud_datacenter" "default" {
  name = "${var.datacenter}"
}

resource "hcloud_volume" "default" {
  count     = "${length(var.instance_type)}"
  name      = "${var.service}-${var.instance_type[count.index]}"
  size      = "${var.volume_size}"
  server_id = "${hcloud_server.default_coreos.*.id[count.index]}"
  automount = "true"
  format    = "xfs"
}

resource "hcloud_server" "default_coreos" {
  count       = "${length(var.instance_type)}"
```

```
  name        = "${var.service}-${var.instance_type[count.index]}"
  image       = "debian-9"
  server_type = "${var.instance_type[count.index]}"
  ssh_keys    = ["${data.hcloud_ssh_key.admin_key.id}"]
  datacenter  = "${data.hcloud_datacenter.default.name}"
  rescue      = "linux64"
}

resource "null_resource" "provisioner" {
  count = "${length(var.instance_type)}"

  triggers {
    public_ip =
"${hcloud_server.default_coreos.*.ipv4_address[count.index]}"
  }

  connection {
    host        =
"${hcloud_server.default_coreos.*.ipv4_address[count.index]}"
    timeout     = "1m"
    agent       = false
    private_key = "${file("${var.ssh_key_file}")}"
  }

  provisioner "file" {
    content     = "${data.ignition_config.default.rendered}"
    destination = "/root/ignition.json"
  }
```

```
  provisioner "remote-exec" {
    script     = "${path.module}/files/install.sh"
    on_failure = "continue"
  }

  provisioner "remote-exec" {
    connection {
      host        =
"${hcloud_server.default_coreos.*.ipv4_address[count.index]}"
      timeout     = "2m"
      agent       = false
      private_key = "${file("${var.ssh_key_file}")}"
      user        = "core"
    }

    inline = "sudo hostnamectl set-hostname
${var.instance_type[count.index]}"
  }

  depends_on = ["hcloud_volume.default"]
}
```

Ansible плейбук для SuperPi тесту

```
---
- name: Run SuperPI container
  docker_container:
    name: superpi-{{ ansible_date_time.hour }}-{{
ansible_date_time.minute }}
    image: bugaga1100/superpi
```

```
    state: started
    env:
      AWS_ACCESS_KEY_ID: key
      AWS_SECRET_ACCESS_KEY: secret
      AWS_DEFAULT_REGION: us-east-1
      SERVER_NAME: "{{ inventory_hostname.split(\".\")[0] }}"

- pause:
    minutes: "{{ superpi_timeout }}"
```

Ansible плейбук для Stress-ng тесту

```
---
- name: Run Stress container
  docker_container:
    name: stress-{{ ansible_date_time.hour }}-{{
ansible_date_time.minute }}
    image: bugaga1100/stress
    state: started
    env:
      AWS_ACCESS_KEY_ID: key
      AWS_SECRET_ACCESS_KEY: secret
      AWS_DEFAULT_REGION: us-east-1
      SERVER_NAME: "{{ inventory_hostname.split(\".\")[0] }}"
      CPU_CORES: "{{ stress_cores }}"
      TIMEOUT: "{{ stress_timeout }}m"

- pause:
    minutes: "{{ stress_timeout }}"
```